File No. 1620-25
Form C26-5876-2

IBM Systems Reference Library

IBM 1620 FORTRAN II Programming System Reference Manual

This manual contains the specifications and operating procedures for Version 2 of the 1620 FORTRAN II Programming System. It should not be used with Version 1 of the subject system. Included are rules for constructing the various types of FORTRAN expressions and statements; procedures for adding subroutines; descriptions of core storage during compilation; and complete operating instructions with details of error messages and restart procedures.

Also described in this manual are the 1620-1443 FORTRAN II and the 1620 FORTRAN II for Automatic Floating Point Programming Systems. The first is a printer-oriented modification of the standard FORTRAN II System, and the latter is designed to compile an object program with in-line arithmetic instructions.

This is a reprint of an earlier edition
incorporating the following Technical Newsletters:

TNL	Pages	Dated
N26-0083	15, 24, 36, 37, 37.1	8/19/64
N26-0112	37, 37.1, 38	3/15/65

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Comments concerning the contents of this publication may be addressed to:
IBM, Product Publications Department, San Jose, Calif. 95114

Contents

Page

IBM 1620 FORTRAN II Programming System 1
Language 1
Arithmetic Modes 2
Constants, Variables and Subscripts 2
Expressions 4
Arithmetic Statements 5
Control Statements 6
Input/Output Statements 10
Specification Statements 12
Functions 16

Compilation Process 22
Pass I 22
Pass II 26
Batch Compilation 27

Object Program 29
Description 29

Page

Execution of the Object Program 30
Error Messages 31
Adding Library Subroutines 34

IBM 1620-1443 FORTRAN II Programming System 36
Language 36
Compiler 36

IBM 1620 FORTRAN II for Automatic Floating Point 37.1

Appendix 38
Supplementary Information 38

Index 42

Preface

FORTRAN II is a coding system with a language that closely resembles the language of mathematics. It is a system designed primarily for scientific and engineering computations. Since this system is essentially problem-oriented rather than machine-oriented, it provides scientists and engineers with a method of communication that is more familiar, easier to learn, and easier to use than actual machine language. In addition, machine language programs produced by the FORTRAN II System are generally as efficient as those which might be written by an experienced programmer.

This manual is a *reference* text for the 1620 FORTRAN II Programming System; it should not be used as a FORTRAN primer. For general information about FORTRAN, refer to the *IBM FORTRAN General Information Manual,* Form F28-8074. For a list of literature applicable to the 1620 Data Processing System, refer to the *IBM 1620 Bibliography,* Form A26-5692.

Machine Configuration and Feature Requirements

The minimum machine configuration and feature requirements needed to use the 1620 FORTRAN II Programming System are as follows:

- IBM 1620 Data Processing System with 40,000 positions of core storage
- IBM 1622 Card Read-Punch
- Automatic Divide
- Indirect Addressing

IBM 1620 FORTRAN II Programming System

The 1620 FORTRAN II Programming System consists of two parts: the language, and the compiler. The language is composed of a number of types of statements that are used to define the problem being solved. The compiler is a program which translates the statements into 1620 machine language.

The problem in FORTRAN II language is called the *source* program; the 1620 machine language translation is called the *object* program.

The FORTRAN II language and the rules for writing source programs are described in the first part of this manual, while the compiler and its related functions are described in the section entitled COMPILATION PROCESS.

Language

A 1620 FORTRAN II source program consists of a number of *statements*. Each statement deals with one aspect of the problem; that is, it may cause data to be fed into the computer, calculations to be performed, decisions to be made, results to be printed, etc.

Some statements do not cause specific computer action, but rather provide information to the compiler program.

The 1620 FORTRAN II statements are arranged into five groups as follows:

- Arithmetic statements, which are used to specify the mathematical calculations to be performed.
- Control statements, which are used to govern the sequence in which the statements will be executed.
- Input/Output statements, which are used to read data into the program, or print or punch the results of the program.
- Specification statements, which are used to provide information about the data that the object program is to process.
- Subprogram statements, which are used to define and use subprograms.

The above statement types are explained in detail later in this manual.

Coding Form

1620 FORTRAN II statements are written on a standard FORTRAN coding form (X28-7327) which is designed to organize the statements into the special format required by the compiler program. All statements and comments of the source program are written on this form. Space is provided at the top of each page for the name of the program, date, etc. This information does not constitute part of the source program and is not punched into cards.

The series of numbers (1, 5, 6, 7, 10, . . . , 72) across the top of the form indicates the card column into which the information is to be punched.

Comments to explain the program are written in columns 2-72 of a line with a C in column 1. A comment line is not processed by the compiler but is listed when the source program cards are listed.

Columns 2 through 5 are used for the *statement number*. Any number from 1 through 9999 may be used as a statement number. Statement numbers are used for cross reference within a program (see explanations of DO and GO TO statements), or merely as a means of identifying statements. No two statements may have the same number. Statements need not be numbered in sequence.

Column 6 of the initial line of a statement must be blank or zero. If a statement is too long to be written on one line, it can be continued on as many as four "continuation lines." Continuation lines are written by placing any character or any number from 1 through 9 (zero allowed only for initial line) in column 6. The normal method is to number the initial line *zero*, the second line *one* (first continuation line), the third line *two*, etc. A statement other than a comment statement may not consist of more than 330 characters (i.e., 5 lines).

The body of a statement is written in columns 7 through 72. Blank columns for the most part are ignored by the compiler and may be used freely to improve the readability of the source program listing.

Columns 73 through 80 are not processed and therefore may contain any identifying information.

FORTRAN Card

To prepare a source program for use, the statement data is transferred from the coding form into cards. After the cards are punched, they should be verified to minimize clerical errors.

Arithmetic Modes

Quantities used in FORTRAN statements may be expressed in either *fixed-point* mode or *floating-point* mode. Numbers expressed as integers (whole numbers) are considered fixed point. Numbers expressed with a decimal point are considered floating point. Thus the numbers 3, 57, and 115 are fixed point numbers, while the numbers 1.72, 35.6, and 1.7772 are floating point numbers.

In FORTRAN II, fixed point and floating point numbers may be used, subject to the rules listed under ARITHMETIC STATEMENTS.

Floating Point Arithmetic

Floating point arithmetic is a technique used to eliminate the complex programming required for correct placement of the decimal point in arithmetic operations. Floating point numbers are represented in a standard format which specifies the location of the decimal point. With this method, quantities which range from minute fractions to large numbers can be handled by the computer. Floating point numbers are expressed as decimal fractions, times a power of ten. For example:

3.14159 is expressed as $.314159 \times 10^1$
4800.0 is expressed as $.48 \times 10^4$
0.0187 is expressed as $.187 \times 10^{-1}$

The numeric part of the floating point number is called the *mantissa;* the power of ten is called the *exponent.*

Constants, Variables and Subscripts

FORTRAN II provides a means of expressing numeric constants, variable quantities, and subscripted variables. The rules for expressing these quantities are quite similar to the rules of ordinary mathematical notation.

Constants

A constant is any number which is used in computation without change from one execution of the program to the next. A constant appears in numeric form in the source statement. For example, in the statement

$$J = 3 + K$$

3 is a constant, since it appears in actual numeric form. Two types of constants may be written in FORTRAN II: fixed point, and floating point.

Fixed Point Constant

A fixed point constant is an integer consisting of 1 to 10 numeric characters (see ARITHMETIC PRECISION). A preceding plus sign is optional for positive numbers. An unsigned constant is assumed to be positive.

EXAMPLES

3
+1
—28987

Floating Point Constant

A floating point constant may be in either of two forms:

1. Any number consisting of 1 to 28 decimal digits with a decimal point at the beginning, at the end, or between two digits (see ARITHMETIC PRECISION). A preceding plus sign is optional for positive numbers. Zeros to the left of the decimal point are permissible.

EXAMPLES

17.
5.0
—.0003
0.0

2. An integral decimal exponent preceded by an E may follow a floating point constant. The magnitude thus expressed must be between the limits of 10^{-100} and 10^{99}, or must be zero.

EXAMPLES

5.0E3	$= (5.0 \times 10^3)$
5.0E + 3	$= (5.0 \times 10^3)$
3.14E	$= (3.14 \times 10^0)$

Variables

A FORTRAN variable is a symbolic name which will assume a value during execution of a program. This value may change either for different executions of the program or at different times within the program. For example, in the statement

$$A = 3.0 + B$$

both A and B are variables. The value of B will be assigned by a preceding statement and may change from time to time; the value of A will change whenever this computation is performed with a new value of B.

As with constants, a variable may be in fixed point or floating point form.

Fixed Point Variables

A fixed point variable is named by using 1 to 6 alphabetic or numeric characters (not special characters) of which the first must be I, J, K, L, M, or N.

EXAMPLES

I
M2
JOBNO1

A fixed point variable can assume any integral value provided the magnitude is less than the maximum size as defined through the use of a control record as stated under ARITHMETIC PRECISION. (If not defined, the maximum size will be 5 decimal positions for fixed point numbers.)

Floating Point Variables

A floating point variable is named by using 1 to 6 alphabetic or numeric characters (not special characters), of which the first is alphabetic but *not* I, J, K, L, M, or N.

EXAMPLES

A
B7
DELTA

A floating point variable may assume any value expressible as a normalized floating point number; i.e., zero or any number between 10^{-100} and 10^{99}. The number of mantissa characters may be from 2 to 28 (see ARITHMETIC PRECISION). If not defined, the maximum size will be 8 characters for the mantissa.

Arithmetic Precision

The precision of the quantities used in the calculation is an important consideration in most types of scientific computation. For example, the computation of 7.19 x 3.14 would not be as precise as 7.19286 x 3.14159.

In the FORTRAN II System, the variable-field length capability of the 1620 is used to allow varying degrees of precision from one program to another. Floating point precision, denoted in this publication as *f*, may be varied from 2 to 28 places; fixed point precision, denoted by *k*, may be varied from 4 to 10 places.

The precision of the values may be specified by the use of a control card which precedes the source program. This card is described in the section entitled SOURCE PROGRAM CONTROL CARD.

Values for *f* and *k* must be the same for subprograms called by the main program.

Subscripts

An *array* is a group of quantities. It is often advantageous to be able to refer to this group by one name and to refer to each individual quantity in this group in terms of its place within the group. For example, assume the following is an array named NEXT:

15
12
18
42
19

If it were desired to refer to the second quantity in the group, the ordinary mathematic notation would be $NEXT_2$. In FORTRAN this becomes

NEXT (2)

The quantity in parentheses is called a subscript. Thus

NEXT (2) has the value of 12
NEXT (4) has the value of 42

The ordinary mathematical notation might be $NEXT_i$, to represent any element of the array NEXT. In FORTRAN, this is written

NEXT (I)

where I equals 1, 2, 3, 4, or 5. A program may also use 2- or 3-dimensional arrays. For example, the following is a 2-dimensional array named MRATE.

	Column 1	Column 2	Column 3
Row 1	14	12	8
Row 2	48	88	4
Row 3	29	25	17
Row 4	1	3	43

To refer to the quantity in Row 4, Column 2, the FORTRAN statement would be written as MRATE (4,2).

The value of MRATE (4,2) is 3.
The value of MRATE (3,3) is 17.

Thus, subscripts are positive fixed point quantities whose values determine the member of the array to which reference is made.

GENERAL FORM

If v represents any fixed point variable and c (or c′) represents any fixed point constant, then a subscript is an expression in one of the following forms.

v, c, v +c, v −c, c*v,
c*v +c′, or c*v −c′

(The symbol * denotes multiplication.)

EXAMPLES

I
3
MU +2
5 * J
5 * J −2

In a subscript the variable itself must *not* be subscripted.

Subscripted Variables

A fixed or floating point variable may be subscripted by enclosing up to three fixed point subscripts in parentheses to the right of the variable.

EXAMPLES

```
A(I)
K(3)
BETA (5*J -2, K +2, L)
```

The commas separating the subscripts are required punctuation. Note that subscript arithmetic may take place as shown in the third example above. For instance, if J is equal to 20, the first subscript will be 98.

The value of a subscript (including the added or subtracted constant, if any) must be greater than zero but not greater than the corresponding array dimension. Each subscripted variable must have the size of its array (i.e., the maximum values which its subscripts can attain) specified in a DIMENSION statement preceding the first appearance of the variable in the source program. (DIMENSION statements are described later.)

Arrangement of Arrays in Storage

Arrays are stored "column-wise," with the first of their subscripts varying most rapidly, and the last varying least rapidly. Arrays which are 1-dimensional are simply stored sequentially. A 2-dimensional array named A would be stored sequentially in the order $A_{1,1}$, $A_{2,1}$, . . . , $A_{M,1}$, $A_{1,2}$, $A_{2,2}$, . . . , $A_{M,N}$. A 3-dimensional array named T would be stored in the order $T_{1,1,1}$, $T_{2,1,1}$, $T_{3,1,1}$, . . . , $T_{M,1,1}$, $T_{1,2,1}$, . . . , $T_{M,N,1}$, $T_{1,1,2}$, $T_{2,1,2}$, . . . , .

The storage of arrays is in ascending order; i.e., the elements are stored sequentially in locations with ascending addresses.

Expressions

An expression in FORTRAN language is any sequence of constants, variables (subscripted or not subscripted), and functions (explained later), separated by operation symbols, commas, and parentheses, which comply with the rules for constructing expressions. *Expressions appear on the right-hand side of arithmetic statements.*

In arithmetic-type operations, the following operation symbols are used:

+	addition
—	subtraction
*	multiplication
/	division
**	exponentiation (i.e., raising to a power)

Rules for Constructing Expressions

Since constants, variables, and subscripted variables may be fixed point or floating point quantities, expressions may contain either fixed point or floating point quantities; however, the two types may appear in the same expression only in certain ways.

1. The simplest expression consists of a single constant, variable, or subscripted variable. If the quantity is an integer quantity, the expression is said to be in the fixed-point mode. If the quantity is a floating point quantity, the expression is said to be in the floating-point mode.

EXPRESSION	TYPE OF QUANTITY	MODE OF EXPRESSION
3	Fixed Point constant	Fixed Point
3.0	Floating Point constant	Floating Point
I	Fixed Point variable	Fixed Point
A	Floating Point variable	Floating Point
I(J)	Fixed Point subscripted variable	Fixed Point
A(J)	Floating Point subscripted variable	Floating Point

In the last example, note that the subscript, which must be a fixed point quantity, does not affect the mode of the expression. *The mode of the expression is determined solely by the mode of the quantity itself.*

2. Exponentiation of a quantity does not affect the mode of the quantity; however, a fixed point quantity may not be given a floating point exponent. The following are valid:

I**J	Fixed Point
A**I	Floating Point
A**B	Floating Point

The following is *not* valid:

I**A	(Violates the rule that a fixed point quantity must not have a floating point exponent)

NOTE: The expression A**B**C is not permitted. It must be written A**(B**C) or (A**B)**C, whichever is intended.

3. Quantities may be preceded by a + or a — or connected by any of the operators (+, —, *, /, **) to form expressions, provided:
   a. No two operators appear consecutively.
   b. Quantities so connected are all of the same mode. (Exception: floating point quantities may have fixed point exponents.)

   The following are valid:

   ```
   —A+B
   B+C—D
   I/J
   K*L
   ```

   The following are *not* valid expressions:

A+ —B	(must be written as A+(—B)
A+I	(variables are of different modes)
3J	(must be written as 3 * J if multiplication is intended)

4. The use of parentheses in forming expressions does not affect the mode of the expression. Thus, A, (A), and (((A))) are all floating point expressions.
5. Parentheses may be used to specify the order of operations in an expression. Where parentheses are omitted, the order is taken to be from *left to right* as follows:

ORDER	SYMBOL	OPERATION
1	**	Exponentiation
2	* and /	Multiplication and Division
3	+ and —	Addition and Subtraction

For example, the expression

A +B*C/D +E**F —G

will be taken to mean

$$A + \frac{B*C}{D} + E^F - G$$

Using parentheses, the expression could be written

(A +B)*C/D +E**F —G

which would be taken to mean

$$\frac{(A+B)*C}{D} + E^F - G$$

A valid expression will be evaluated when the object program is executed. An invalid expression may result in an error message from the FORTRAN II compiler or may result in inaccurate object program results.

Arithmetic Statements

GENERAL FORM

A = B

where A is a *variable* (subscripted or not subscripted) and B represents an *expression.*

EXAMPLE

```
Q = K +1
A(I) = 2(I) + SINF (C(I))
```

The numeric calculations to be performed in the object program are defined by arithmetic statements. FORTRAN arithmetic statements closely resemble conventional arithmetic formulas. They contain a variable to be computed, followed by an equal (=) sign, followed by an arithmetic expression. In FORTRAN language, the equal sign means *"is to be replaced by"* rather than "is equivalent to." For example, the arithmetic statement

Y = N —LIMIT (J —2)

means that the value in the storage area assigned to Y is to be replaced by the value of N —LIMIT (J —2). The equal sign description can be emphasized more with the example of

I = I +1

which means that the variable I *is to be replaced* with its old value plus one.

The result of the expression is stored in fixed point form if the variable to the left of the equal sign is a fixed point variable; it is stored in floating point form if the variable is a floating point variable.

If the variable to the left is in fixed point form and the expression to the right is in floating point form, the result is first computed in floating point, then truncated (the fractional value is dropped) and converted to a fixed point number. Thus, if the result of an expression is 3.872, the fixed point number stored is 3, not 4. Likewise, the statement

J = A/B

produces a result of 1 if the value of A is 7, and the value of B is 4.

If the variable to the left is in floating point form and the expression to the right is in fixed point form, the expression will be computed in fixed point and then converted to floating point before it is stored as the new value of the variable.

EXAMPLES	MEANING
A = B	Store the value of B in A.
I = B	Truncate B to an integer, convert to fixed point, and store in I.
A = I	Convert I to floating point, and store in A.
A = 3.0*B	Replace A with 3 times B.
A = I*B	Not permitted. The expression is *mixed;* i.e., contains both fixed point and floating point variables.
A = 3 * B	Not permitted. The expression is *mixed.*

Control Statements

The second class of FORTRAN II statements is comprised of control statements that enable the programmer to state the flow of the program. Normally, statements may be thought of as being executed sequentially; that is, after one statement has been executed, the statement immediately following is executed. However, it is often undesirable to proceed in this manner. The following statements may be used to alter the sequence of a program.

GO TO Statement (Unconditional)

This statement interrupts the sequential execution of statements, and specifies the number of the next statement to be performed.

GENERAL FORM

GO TO n

where *n* is a statement number.

EXAMPLES

GO TO 1009
GO TO 3

Computed GO TO

This statement also indicates the statement that is to be executed next. However, the statement number that the program is transferred to can be altered during the program.

GENERAL FORM

GO TO $(n_1, n_2, \ldots, n_m)$, i

where n_1, n_2 . . . , n_m are statement numbers and *i* is a non-subscripted fixed point variable.

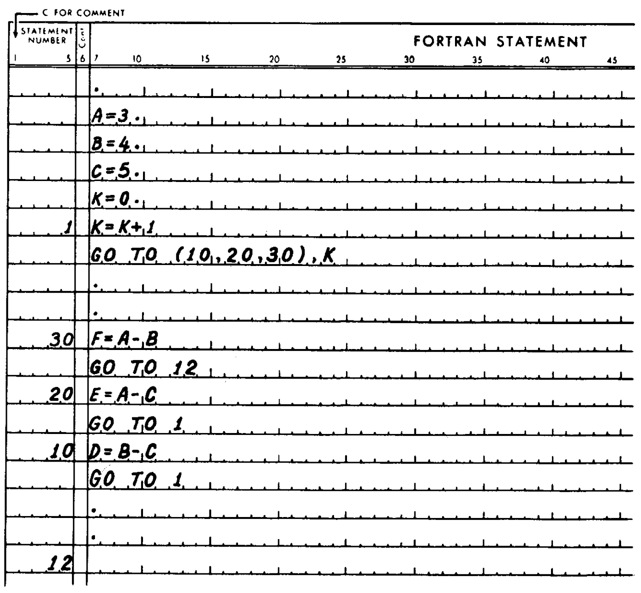
The parentheses enclosing the statement numbers, the commas separating the statement numbers, and the comma following the right parenthesis are all required punctuation.

This statement causes transfer of control to the first, second, third, etc., statement in the list depending on whether the value of *i* is 1, 2, 3, etc.

The variable i must never have a value greater than the number of items in the list.

EXAMPLES	MEANING
GO TO (3, 4, 5), L	If L is 1, transfer to statement 3. If L is 2, transfer to statement 4. If L is 3, transfer to statement 5.
GO TO (4, 4, 5, 2), J	If J is *1 or 2,* transfer to statement 4. If J is 3, transfer to statement 5. If J is 4, transfer to statement 2.

Further examples of the Computed GO TO and the Unconditional GO TO statements are illustrated below:


C FOR COMMENT
STATEMENT NUMBER | FORTRAN STATEMENT

```
      .
      A=3.
      B=4.
      C=5.
      K=0.
    1 K=K+1
      GO TO (10,20,30),K
      .
      .
   30 F=A-B
      GO TO 12
   20 E=A-C
      GO TO 1
   10 D=B-C
      GO TO 1
      .
      .
   12
```

In the example, D, E, and F are computed in that order, and the program is transferred to statement 12. This is a simplified example and if these were the only computations in the program, the programmer would simply list the arithmetic statements to compute D, E, and F in the desired order without using the Computed GO TO statement.

IF Statement

This statement permits the programmer to change the sequence of statement execution depending upon the value of the arithmetic expression.

GENERAL FORM

IF $(a) n_1, n_2, n_3$

where a is an expression, and n_1, n_2, n_3 are statement numbers.

The expression must be enclosed in parentheses and the statement numbers must be separated by commas. The expression may be in either fixed or floating point mode.

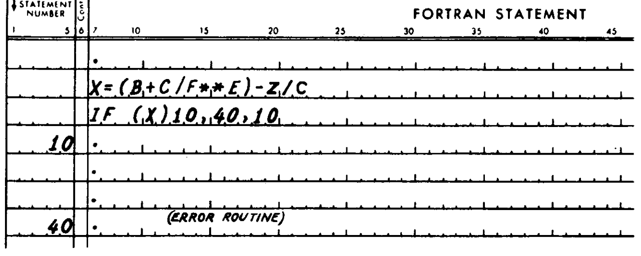
Control is transferred to statement number n_1, n_2, n_3 depending on whether the value of a is *less than, equal to,* or *greater than* zero, respectively.

EXAMPLE

IF (A —B) 10, 5, 7

which means "If the value of A minus B is less than zero, transfer to statement 10. If the value of A minus B is equal to zero, transfer to statement 5. If the value of A minus B is greater than zero, transfer to statement 7."

Suppose a value, X, is being computed. Whenever this value is negative or positive, it is desired to proceed with the program. Whenever the value is zero, an error routine is to be followed. This may be coded as:



```
C FOR COMMENT
STATEMENT
 NUMBER   Cont                         FORTRAN STATEMENT
1     5  6  7  10   15   20   25   30   35   40   45
            .
            X=(B+C/F**E)-Z/C
            IF (X)10,40,10
   10       .
            .
            .
   40       .        (ERROR ROUTINE)
```

IF (SENSE SWITCH) Statement

This statement permits the program to transfer to a particular statement depending on the setting of any one of the four Console Program switches.

GENERAL FORM

IF (SENSE SWITCH i) n_1, n_2

where i is the number of one of the Console Program switches, and n_1, n_2 are statement numbers.

The parentheses enclosing the words SENSE SWITCH, and the commas separating the statement numbers are required punctuation.

The program transfers to statement number n_1 when the designated Program switch is on, and to statement number n_2 when it is off.

EXAMPLE

IF (SENSE SWITCH 3) 14, 10

which means, "If Sense Switch 3 is on, transfer to statement 14; otherwise, transfer to statement 10."

DO Statement

GENERAL FORM

DO n i = m_1, m_2

or

DO n i = m_1, m_2, m_3

where n is a statement number, i is a non-subscripted fixed point variable, and m_1, m_2 and m_3 (none of which may be zero) are either unsigned fixed point constants or non-subscripted fixed point variables. If *m_3 is not stated, it is understood to be 1.*

EXAMPLES

DO 30 J = 1, 10

DO 30 J = 1, K, 3

The DO statement is a command to repeatedly execute the statements that follow, up to and including the statement with statement number n. In other words, a DO statement forms a program loop.

The statements are executed with $i = m_1$ the first time; for each succeeding execution, i is increased by m_3. After the statements have been executed with i equal to m_2 (or as near as possible *without exceeding* m_2), control passes to the statement following the last statement in the range of the DO.

DO Range

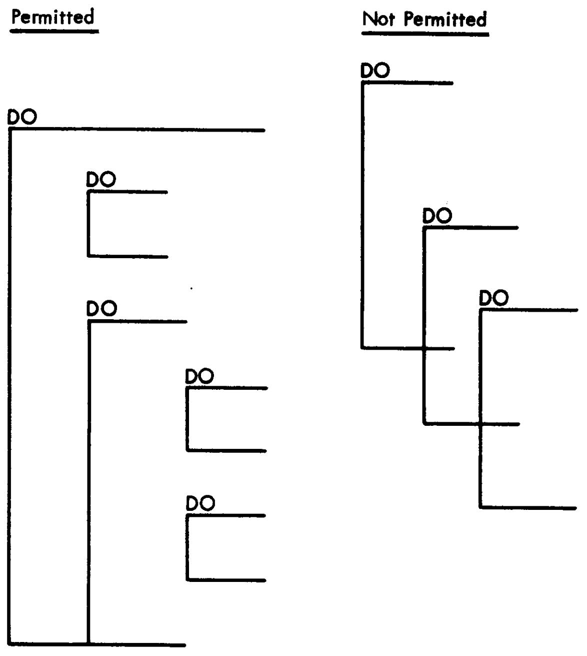
The range of a DO is that set of statements which are executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO, up to and including the statement numbered *n*.

DO's Within DO's

There may be other DO statements among the statements in the range of a DO. When this is so, the following rule must be observed.

If the range of a DO includes one or more other DO's, then all of the statements in the range of the latter must also be in the range of the former.

A set of DO's satisfying this rule is called a "nest of DO's." This rule is illustrated in the drawing below. (Brackets are used to illustrate the range of a DO).



DO Index

The index of a DO statement is the fixed point variable *i*, which is controlled by the DO in such a way that its value begins at m_1, and is increased each time by m_3, up to, *but not including* the value which exceeds m_2. Throughout the range, the *i*-value is available for computation, either as an ordinary fixed point variable or as the variable of a subscript. After the last execution of the range, the DO is said to be "satisfied."

Suppose for example, that control has reached statement 10 of the program:

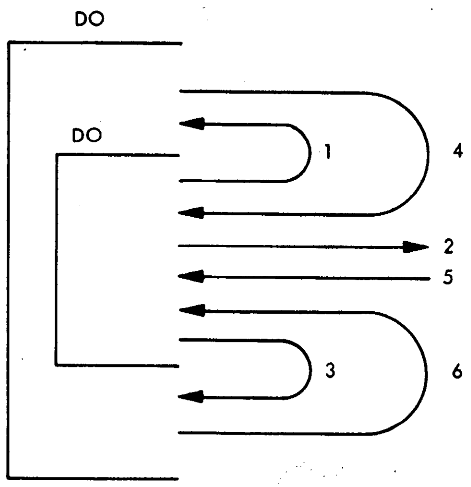
```
10  DO 11 I = 1, 10
11  A (I) = I * N (I)
12
```

The range of the DO is statement 11, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1 * N(1) is computed, converted to floating point and stored in location A(1). Since statement 11 is the last statement in the range of the DO, and the DO is unsatisfied, I is increased to 2 and control returns to the beginning of the range, (which is statement 11). The value of 2 * N(2) is then computed and stored in location A(2). The process continues until statement 11 has been executed with I = 10. Since the DO is then satisfied ($m_1 = m_2$), control passes to statement 12.

Transfer of Control Within a DO

Transfers of control from and into the range of a DO are subject to the following rule:

No transfer is permitted into the range of any DO from outside its range. Thus, 1, 2, and 3 are allowable transfers in the drawing below, but 4, 5, and 6 are not.



Preservation of Index Values. When control leaves the range of a DO in the ordinary way (i.e., when the DO becomes satisfied and control passes on to the next statement after the range), the exit is said to be a normal exit. After a normal exit from a DO occurs, the value of the index controlled by that DO is not defined, and the index cannot be used again until it is redefined. However, if the exit occurs by virtue of a transfer out of the range, the current value of the index remains available for any subsequent use. If the exit occurs because of a transfer which is in the ranges of several DO's, the current values of all indexes controlled by those DO's are preserved for any subsequent use.

Exits. When a CALL statement (see CALL STATEMENT) is executed in the range of a DO, care must be taken that the called subprogram does not alter the DO index or indexing parameters. This applies as well when a FORTRAN function is called for in the range of a DO.

Restrictions on Statements in the Range of a DO. A statement which redefines the value of the index or of any of the indexing parameters (m's) is the only type of statement not permitted in the range of a DO. In other words, the indexing of a DO loop must be completely set before the range is entered. The first statement in the range of a DO must not be a non-executable statement, such as END, CONTINUE, and FORMAT statements. Also, a DO loop cannot end with a transfer statement.

CONTINUE Statement

CONTINUE is a dummy statement which results in no instructions in the object program. It is most frequently used as the last statement in the range of a DO to provide a transfer address for IF and GO TO statements that are intended to begin another repetition of the DO loop.

EXAMPLE

```
CONTINUE
```

As an example of a program which requires a CONTINUE, consider the table search:

```
   .
   .
   .
10 DO 12 I = 1, 100
   IF (ARG -VALUE (I)) 12, 20, 12
12 CONTINUE
13 ...
   .
   .
```

This program causes a scan of the 100-entry VALUE table until it finds an entry that equals the value of the variable ARG, whereupon it exits to statement 20 with the value of I available for fixed point use; if no entry in the table equals the value of ARG, a normal exit occurs to the statement (13) following the CONTINUE.

PAUSE Statement

GENERAL FORM

PAUSE or PAUSE n

where *n* is an unsigned fixed point constant

EXAMPLES

```
PAUSE
PAUSE 33333
```

This statement halts the machine. Pressing the Start key causes the program to resume execution of the object program with the next statement. In a PAUSE *n* statement, where *n* is a 5-digit number within the range of valid 1620 addresses, the *n* can be displayed on the 1620 console in OR-2.

STOP Statement

GENERAL FORM

STOP or STOP n

where *n* is an unsigned fixed point constant.

EXAMPLES

```
STOP
STOP 33333
```

This statement causes a halt in such a way that pressing the Start key has no effect. Therefore, in contrast to PAUSE, this statement is used where a terminal, rather than a temporary stop, is desired. When this statement is executed, the message "STOP" is typed on the console typewriter, and *n* can be displayed as in a PAUSE *n* statement.

END Statement

GENERAL FORM

END or END (I_1, I_2, I_3, I_4, I_5)

where I is 0, 1, or 2.

EXAMPLES

```
END
END (1, 2, 0, 1, 1)
```

This statement differs from the previous control statements in that it does not affect the flow of control in the object program being compiled. It applies to the FORTRAN II compiler during compilation. An END statement will generate a halt and branch in the object program. The statement END (I_1, I_2, I_3, I_4, I_5) is acceptable; however, the I's specified are meaningless in 1620 FORTRAN II.

The END statement must be the last statement (physically) of the source program.

Input/Output Statements

Input statements are used to read data into core storage and output statements are used to print or punch data. The READ, ACCEPT, ACCEPT TAPE, PUNCH, PUNCH TAPE, PRINT, and TYPE statements require the use of the FORMAT statement which is described in the section entitled SPECIFICATION STATEMENTS.

Specifying Lists of Quantities

The input/output statements that call for transmission of data must include an ordered list of the quantities to be transmitted. The listed order must be the same as the order in which the words of information exist (for input), or the desired order for the output.

The formation and meaning of a list is best described by an example. Assume that the value of K has been previously defined.

```
A, B(3), (C(I), D(I,K), I = 1, 10),
((E(I,J), I = 1, 10, 2), F(J,3), J = 1, K)
```

If this list is used with an output statement, the information will be written on the output medium in this order:

```
A, B(3), C(1), D(1, K), C(2), D(2, K), ..,
C(10), D(10, K), E(1, 1), E(3,1), . . . , E (9,1),
F(1, 3),
E(1, 2), E(3, 2), . . . , E(9, 2), F(2, 3),
. . . ,
E(1, K), E(3, K), . . . , E(9, K), F(K, 3)
```

Similarly, if this list is used with an input statement, the successive values, as they are read from the external medium, are placed into core storage in the indicated order. The list reads from left to right with repetition for variables enclosed within parentheses. Only variables, not constants, may be listed.

If such a list is used, the execution is exactly that of a DO loop. It is as though each opening parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching closing parenthesis, and with the DO range extending up to that indexing information. The order of the above list can thus be considered the equivalent of the following "program":

```
1. OUTPUT A
2. OUTPUT B(3)
3. DO 5 I = 1, 10
4. OUTPUT C(I)
5. OUTPUT D(I, K)
6. DO 9 J = 1, K
7. DO 8 I = 1, 10, 2
8. OUTPUT  E (I, J)
9. OUTPUT  F (J, 3)
```

Note that indexing information, as in DO's, consists of three constants or fixed point variables, and that the last of these may be omitted, in which case it is assumed to be 1.

For a list of the form K, A(K) or of the form K, (A(I), I = 1, K), where an index or indexing parameter itself appears earlier in the list of an input statement, the indexing will be carried out with the newly read-in value.

Input/Output in Matrix Form

As outlined in a previous section, FORTRAN II treats variables according to conventional matrix practice. Thus, the input/output statement,

```
READ 1, ( ( A (I, J), I = 1, 2), J = 1, 3)
```

causes the reading of I times J (in this case, 2 times 3) items of information. The data items are read into storage in the same order as they are found on the input medium.

Input/Output of Entire Matrices

When input or output of an entire matrix is desired, an abbreviated notation may be used for the list of the input/output statement; only the name of the array need be given and the indexing information may be omitted.

Thus, if A has previously been listed in a DIMENSION statement, the statement,

```
READ 1, A
```

is sufficient to read in all elements of the array. The elements of the array are stored in successively higher storage locations. (If A has not previously appeared in a DIMENSION statement, only the first element would be read in.)

Automatic Fix/Float

During execution of input/output statements, it is permissible to read a fixed point argument into a floating point field or a floating point argument into a fixed point field, and to write from a floating point field in a fixed point format or from a fixed point field in floating point format. During reading, the format specification dictates the data conversion, and the list designation controls the mode of storing the argument. During writing, the format specification dictates the mode of the field which is printed or punched.

READ Statement

The READ statement is used to read data into core storage from the 1622 Card Read-Punch.

GENERAL FORM

READ n, List

where *n* is the statement number of a FORMAT statement and List is a list of the quantities to be read.

EXAMPLES

READ 8, A, B, C
READ 211, VOLT (I), OHM (J)

The READ statement causes data to be read from a card and the quantities from the card to become the values of the variables named in the list. Successive cards are read until the complete list has been "satisfied"; i.e., all data items have been read, converted, and stored in the locations specified by the list of the READ statement. The FORMAT statement to which the READ refers, describes the arrangement of information on the cards and the type of conversion to be made.

ACCEPT TAPE Statement

The ACCEPT TAPE statement is used to cause data to be read into core storage from the 1621 Paper Tape Reader.

GENERAL FORM

ACCEPT TAPE n, List

where *n* is the statement number of a FORMAT statement, and List is described under INPUT/OUTPUT STATEMENTS.

EXAMPLE

ACCEPT TAPE 30, K, A (J)

The ACCEPT TAPE statement causes the object program to read information from the paper tape reader. Record after record is brought in, in accordance with the FORMAT statement, until the complete list has been satisfied.

ACCEPT Statement

The ACCEPT statement is used to allow data to be read in from the console typewriter.

GENERAL FORM

ACCEPT n, List

where *n* is the statement number of a FORMAT statement, and List is as described under INPUT/OUTPUT STATEMENTS.

EXAMPLE

ACCEPT 20, A, B, C, D (3)

The ACCEPT statement causes the object program to return the carriage of the console typewriter to await the entry of data. The information is entered in accordance with the FORMAT statement until the complete list has been satisfied.

PUNCH Statement

The PUNCH statement is used to cause data to be punched out in cards by the 1622 Card Read-Punch.

GENERAL FORM

PUNCH n, List

where *n* is the statement number of a FORMAT statement, and List is as described under INPUT/OUTPUT STATEMENTS.

EXAMPLE

PUNCH 40, (A (J), J = 1, 10)

The PUNCH statement causes the object program to punch cards in accordance with the FORMAT statement until the complete list has been satisfied.

PRINT and TYPE Statements

The PRINT statement and the TYPE statement are used to type out data on the console typewriter.

GENERAL FORM

PRINT n, List
TYPE n, List

where *n* is the statement number of a FORMAT statement and List is as described under INPUT/OUTPUT STATEMENTS.

EXAMPLE

PRINT 2, (A (J), J = 1, 10)

The PRINT and TYPE statements cause output data to be typed on the console typewriter. A carriage return occurs, and successive lines are typed in accordance with the FORMAT statement until the complete list has been satisfied.

PUNCH TAPE Statement

The PUNCH TAPE statement is used to cause data to be punched by the paper tape punch.

GENERAL FORM

```
PUNCH TAPE n, List
```

where *n* is the statement number of a FORMAT statement, and List is as described under INPUT/OUTPUT STATEMENTS.

EXAMPLE

```
PUNCH TAPE 25, ( A (J), J = 1, 10)
```

The PUNCH TAPE statement causes information to be punched by the paper tape punch.

Successive records are punched in accordance with the FORMAT statement until the complete list has been satisfied.

Specification Statements

The Specification statements supply necessary information to the FORTRAN compiler, or information to increase program efficiency. No executable instructions are created in the object program as a result of a Specification statement.

DIMENSION Statement

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program.

GENERAL FORM

```
DIMENSION v, v, v . . .
```

where each v is the name of a variable, subscripted with 1, 2, or 3 unsigned fixed point constants. Any number of v's may be given.

EXAMPLE

```
DIMENSION A(10), B(5, 15), CVAL (3, 4, 5)
```

Each variable which appears in subscripted form in a program or subprogram must appear in a DIMENSION statement of that program or subprogram. The DIMENSION statement must precede the first appearance of that variable. The DIMENSION statement lists the maximum dimensions of arrays. In the object program, references to these arrays must never exceed the specified dimensions.

The above example indicates that B is a 2-dimensional array for which the subscripts never exceed 5 and 15. The DIMENSION statement, therefore, causes 75 (i.e., 5 x 15) fields to be set aside for the B array.

A single DIMENSION statement may specify the dimensions of a number of arrays. The maximum number is limited by the number of continuation cards permitted. A program must not contain a DIMENSION statement which includes the name of the program itself, or any program which it calls. If any of the subscripts in a DIMENSION statement exceeds 9999, an error will be indicated.

EQUIVALENCE Statement

The EQUIVALENCE statement provides one method of controlling the allocation of data storage in the object program.

GENERAL FORM

```
EQUIVALENCE (a, b, c, . . .), (d, e, f, . . . ) , . . .
```

where a, b, c, d, e, f, . . . are variables that may be subscripted with constants only.

EXAMPLE

```
EQUIVALENCE ( A, B(1), C(5) ), ( D (17),E(3))
```

When the logic of the program permits, the number of storage locations used can be reduced by causing locations to be shared by two or more variables. The EQUIVALENCE statement should not be used to obtain mathematical equality between two or more elements. If fixed point and floating point variables are equivalenced, their word lengths must be the same; i.e., $f + 2$ must equal k.

An EQUIVALENCE statement may be placed anywhere in the source program except as the first statement in the range of a DO. Each set of parentheses in the statement list encloses the names of two or more quantities which are to be stored in the same locations during execution of the object program; any number of equivalences may be given.

In an EQUIVALENCE statement, a term such as C(p) can be defined for $p>0$ to mean the *p*th location of the C array. For example, C(5) would be the fifth location in the C array. If p is not specified, it is understood to be 1.

Thus, the example indicates that the A, B, and C arrays are to be assigned storage locations such that the elements A(1), B(1), *and* C(5) occupy the same location. In addition, it specifies that D (17) and E(3) are to share the same location.

Quantities or arrays which are not mentioned in an EQUIVALENCE statement are assigned unique locations.

Locations can be shared only among variables, not among constants.

The sharing of storage locations cannot be planned safely without a knowledge of which FORTRAN II statements, when executed in the object program, will cause a new value to be stored in a location. There are five such FORTRAN II statements:

1. Execution of an Arithmetic statement stores a new value of the variable for the left-hand side of the formula.
2. Execution of a DO statement stores a new indexing value.
3. Execution of a READ, ACCEPT, or ACCEPT TAPE statement stores new values for the variables mentioned in the statement list.

COMMON Statement

Variables, including arrays, appearing in COMMON statements are assigned to specific storage locations. Storage is assigned separately for each program compiled.

GENERAL FORM

```
COMMON A, B . . .
```

where A, B . . . are the names of variables and *non*-subscripted array names.

EXAMPLE

```
COMMON X, ANGLE, MATA, MATB
```

The COMMON storage area may be shared by a program and its subprograms. In this way, the COMMON statement enables a data storage area to be shared between programs in a way analogous to that by which the EQUIVALENCE statement permits data storage-sharing within a single program. Where the logic of the programs permits, this can result in a large saving of storage space.

Array names appearing in the COMMON statement must previously have appeared in a DIMENSION statement in the same program.

The COMMON storage area is located at the high end of core storage, starting with address 19999, 39999 or 59999. Variables in a COMMON statement are assigned storage locations in descending sequence. For example:

```
COMMON A, B, C
```

With $f = 10$, A, B, and C would be stored in locations 19999, 19987, and 19975, and similarly for 40,000 or 60,000 positions. If C is dimensioned as C(10), then 19975 is the address of C(10), which is the last element in the array, and 19867 is the address of C(1).

The COMMON statement takes precedence over the EQUIVALENCE statement. Due to the complex interaction of these two statements, the programmer must adhere to the following two rules:

1. Variables which are to be placed in COMMON storage must be assigned prior to any EQUIVALENCE statement containing these variables. For example,

   ```
   COMMON A
   EQUIVALENCE (A, B, C)
   ```

   The order in which the variables appear in the EQUIVALENCE statement is irrelevant and Rule 1 applies if the COMMON variable is B or C.
2. Within an EQUIVALENCE list there may be no more than one variable which previously has been:
   a. equivalenced, or
   b. placed in COMMON.

The following sequence of statements is invalid.

```
EQUIVALENCE (A, B, C)
EQUIVALENCE (X, Y, Z)
EQUIVALENCE (A, Z)      Violates (a) above
COMMON      D
EQUIVALENCE (D, X, P)   Violates combina-
                        tion of (a) and (b)
```

The sharing of storage locations desired in the above statements can be achieved by writing the statements as follows.

```
COMMON      D
EQUIVALENCE (D, X, P)
EQUIVALENCE (A, B, C, X)
EQUIVALENCE (X, Y, Z)
```

or

```
COMMON      D
EQUIVALENCE (D, A, P, B, C, X, Y, Z)
```

A diagnostic error message results if either Rule 1 or 2 is violated.

Arguments in Common Storage

COMMON statements may be used as a medium for transmitting arguments from the calling program to the called FORTRAN function or SUBROUTINE subprogram. In this way, they are *implicitly*, rather than *explicitly* transmitted as when listed in the parentheses following the subprogram name.

To obtain implicit arguments, it is necessary to have only the corresponding variables in the two programs occupy the same location. This can be accomplished by having them occupy corresponding positions in

COMMON statements of the two programs. For example, (A, B, C) and (E, F, G) become implicit arguments when the calling program contains the statement COMMON A, B, C, and the called subroutine contains the statement, COMMON E, F, G.

NOTES:

1. To force correspondence in storage locations between two variables in different programs, which otherwise would occupy different relative positions in COMMON storage, it is valid to place dummy variable names in a COMMON statement. These dummy names, which may be dimensioned, will cause reservation of the space necessary for correspondence.
2. While implicit arguments can take the place of all arguments in CALL-type subroutines, there must be at least one explicit argument in a FORTRAN function. Here, too, a dummy variable may be used for convenience.

   When one variable is equivalenced to a second variable which appears in a COMMON statement, the first variable is also located in COMMON storage.

FORMAT Statement

The FORMAT statement is used to describe the format of data being transmitted to and from the typewriter, card, or paper tape units.

GENERAL FORM

FORMAT $(s_1, \ldots, s_n)$

where s_i is a format specification. The FORMAT specifications must be separated by commas, slashes, or left parentheses.

EXAMPLE

FORMAT (I2/ (E12.4, F10.4))

The Input/Output statements, in addition to the list of quantities to be transmitted, contain the statement number of a FORMAT statement describing the information format to be used. The FORMAT statement also specifies the type of conversion to be performed between the internal machine language and the external notation. FORMAT statements are not executable; their function is merely to supply information to the object program. Therefore, they may be placed anywhere in the source program (except as the first statement in the range of a DO).

For the sake of clarity, examples given in this section are for typing on the console typewriter. However, the description is valid for any input/output unit simply by generalizing the concept of "typewritten line" to that of the unit record in the selected input/output unit. Thus, a unit record may be:

1. A typewritten line with a maximum of 87 characters.
2. A punched card with a maximum of 80 characters.
3. A paper tape record with a maximum of 87 characters. (The input record length may be variable up to 87; the output record length is fixed at 87).

Numeric Fields

Three forms of conversion for numeric data are available:

FROM/TO INTERNAL	TYPE	TO/FROM EXTERNAL
Floating point variable	E	Floating point decimal number
Floating point variable	F	Fixed point decimal number
Fixed point variable	I	Integer

These types of conversion are specified in the forms:

Ew.d, Fw.d, and Iw.

where w and d are unsigned fixed point constants.

Format specifications are used to describe input and output formats. The format is specified by giving, from left to right, beginning with the first character of the record:

1. The control character (E, F, or I) for the field.
2. The width (w) of the field. The width specified may be greater than required to provide spacing between numbers.
3. For E- and F-type conversions, the number of decimal positions (d) (of the field) which appear to the *right* of the decimal point. A maximum of *f* digits to the right of the decimal point is allowed for the output.

Specifications for successive fields are separated by commas. No format specification that provides for more characters than the input/output unit record should be given. Thus, a FORMAT statement for typewritten output should not provide for more than 87 characters per line including blanks.

Example: The statement, FORMAT (I2, E12.4, F10.4) might cause typing of the line:

$$\overbrace{\text{b7}}^{\text{I2}}\ \overbrace{-92.3100\text{E}+00}^{\text{E12.4}}\overbrace{\text{bbbb}-.0076}^{\text{F10.4}}$$

(In these examples, b is included to indicate blank spaces.)

Alphameric Fields

FORTRAN II provides a method by which alphameric information may be read or written.

The specification for this purpose, wH, is followed in the FORMAT statement by w alphameric characters. For example:

24H THIS IS ALPHAMERIC DATA

Note that blanks are considered alphameric characters and must be included as part of the count w.

Information handled with the H specification is not given a name and may not be referred to or manipulated in storage in any way.

The effect of wH depends on whether it is used with input or output.

1. *Input.* w characters are extracted from the input record to replace the w characters included with the specifications.
2. *Output.* The w characters following the specifications, or the characters which replaced them, are written as part of the output record.

Example: The statement, FORMAT (3HXY = F8.3) could produce any of the following lines:

```
XY = b –93.210
XY = b999.999
XY = bb28.768
```

Another alphameric specification, Aw causes w alphameric characters to be read into or written from a variable or array name. Since each alphameric character is represented in core storage by two decimal digits, w must be less than, or equal to, the largest whole number resulting from $k/2$ or $f/2$, depending on whether the variable or array name is fixed or floating. If k or f is odd, a zero will be supplied as the least significant digit for the field in core storage. To facilitate manipulation of alphameric fields which are stored as floating point numbers, the numbers will have zeros as an exponent. This will have no effect on input/output. However, if the first character in a field is a blank, decimal point, or close parenthesis, the field will be treated as a zero in the floating point arithmetic subroutines.

Blank Fields

Blank characters may be provided in an output record, and characters of an input record may be skipped, by means of the specification wX where $0 < w \leq 87$ (w is the number of blanks provided or characters skipped). When the specification is used with an input record, w characters are considered to be blank (regardless of what they actually are) and are skipped over.

Repetition of Field Format

It may be desirable to print n successive fields within one record, in the same fashion. This may be specified by giving n (where n is an unsigned fixed point constant which must be ≤ 99) before E, F, I, or A. Thus, the statement, FORMAT (I2, 3E12.4) might result in:

```
27 –92.3100E + 00b75.8000E –02b55.3600E –02
```

Repetition of Groups

A limited, one-level, parenthetical expression is permitted in order to enable repetition of data fields according to certain format specifications within a longer FORMAT statement specification. Thus, FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F.10.6, E10.2, F10.6, E10.2, I4). The number of repetitions is limited to a maximum of 99.

Scale Factors

The E-type specification implies a scale factor. Therefore E16.8 for an output field will result in the printing or punching of a maximum of ten significant digits in the form (–)XX.XXXXXXXXE(–)XX. A maximum of f digits can be placed to the right of the decimal point if the d specification is greater than f. In this case d $-f$ low order zeros will be inserted to satisfy the d specification. The following guide may be used when working with E-type specifications.

1. If f (floating point precision) $\leq$ w –6, then f significant digits will be printed or punched.
2. If f $>$w –6, then w –6 significant digits will be printed or punched.

For example, if $f = 10$ and the floating point number is stored as $\bar{1}23456789\bar{1}\bar{3}\bar{5}$, it will be printed as –12.34567891E –37, according to specification E16.8.

The F-type specification also implies a scale factor. Therefore, F16.8 for an output field will result in the printing or punching of a maximum of fourteen significant digits in the form (–)XXXXXX.XXXXXXXX. However, a maximum of f digits will be placed to the right of the decimal point and the result will be right-justified in the output field. If f is larger than w –2, only w –2 digits will appear in the output.

The X specification should be used to space fields in the E-type format. In the statement

E16.8, 1X, E16.8, 1X, E16.8

a space will be provided between adjacent fields.

A field read according to the E-type format need not have the exponent E(–)XX; i.e., it may actually take the same form as the F-type format.

The P-scale factor may be used in a specification, but it will be ignored by the FORTRAN II compiler.

Multiple Record Formats

To deal with a block of more than one typewritten line, a FORMAT specification may have several different one-line formats, separated by a slash (/) to indicate the beginning of a new line. Thus, FORMAT (3F9.2, 2F10.4/6E14.5) specifies a multiline typewritten block in which line 1 has format 3F9.2 and 2F10.4, and line 2 has format 6E14.5

If a multiple-line format is desired, such that the first two lines are typed according to a special format and all remaining lines are typed according to another format, the last line specification should be enclosed in a second set of parentheses; e.g., FORMAT (I2, 3E12.4/2F10.3, 3F9.4/(5F12.4)). If data items remain to be transmitted after the last line format specification has been completely satisfied, the format repeats from the last left parenthesis.

As these examples show, both the slash and the closing parenthesis of the FORMAT statement indicate the termination of a record.

Blank lines may be introduced into a multiline FORMAT statement by listing consecutive slashes.

FORMAT and Input/Output Statement Lists

The FORMAT statement indicates, among other things, the maximum size of each record to be transmitted. In this connection it must be remembered that the FORMAT statement is used in conjunction with the list of some particular input/output statement, except when a FORMAT statement consists entirely of alphameric fields. When the FORMAT statement is used with the list, control in the object program transfers back and forth between the list (which specifies whether data remains to be transmitted) and the FORMAT statement (which gives the specifications for transmission of that data).

Ending a FORMAT Statement

During input/output of data, the object program scans the FORMAT statement to which the relevant input/output statement refers. When a specification for a numeric field is found and list items remain to be transmitted, input/output takes place according to the specification, and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, a numeric input/output operation will be brought to an end when a specification for a numeric field or the end of the FORMAT statement is encountered, and there are no items remaining in the list.

Data Input to the Object Program

Input data to be read when the object program is executed must be in essentially the same format as given in the previous examples. Thus, a card to be read according to FORMAT (I2, E12.4, F10.4) might be punched:

27b −0.9321Eb02bbb −0.0076

Within each field, all information must appear at the extreme right. Plus signs may be omitted or indicated by a b (blank) or +. Blanks in numeric fields are regarded as zeros, but zeros may not be substituted for blanks. For example, a sign cannot be preceded by zeros. Numbers for E-type and F-type conversion may contain any number of digits, but only the high-order *f* digits of accuracy are retained. Numbers for I-type conversion may not contain more than *k* significant digits.

To permit economy in punching, certain relaxations in input data format are permitted.

1. Numbers for E-type conversion need not have four columns devoted to the exponent field. The start of the exponent field must be marked by an E, or if the E is omitted, by a + or − (not by a blank). Thus E2,E02, +2, +02, Eb02, and E +02 are all permissible exponent fields. Blanks are not permitted between characters in the exponent field except that a blank may be used in place of a plus sign. Numbers for E-type conversion must be right-justified in the data record field.
2. Numbers for E-type or F-type conversion need not have their decimal points punched. If not punched, the FORMAT specification will supply them; for example, the number −09321 +2 with the specification E12.4 will be treated as though the decimal point has been punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the indicated position in the FORMAT specification.

Functions

Functions in the 1620 FORTRAN II System are divided into three types. These are (1) Library functions, (2) Arithmetic Statement functions, and (3) FORTRAN (subprogram) functions.

Library Functions

Seven library (closed) functions are included in the FORTRAN II System, as distributed. These functions are:

NAME	TYPE OF FUNCTION
LOGF	Natural Logarithm
SINF	Trigonometric Sine
COSF	Trigonometric Cosine
EXPF	Exponential
SQRTF	Square Root
ATANF	Arctangent
ABSF	Absolute Value Function

Library functions are prewritten and exist in prepared card decks. These functions constitute "closed" subroutines; that is, instead of appearing in the object program every time they are referred to in the source program, they appear only once.

Library functions can be "called," or used, by including the name of the function in an arithmetic expression. The name is followed by an argument enclosed in parentheses. The argument can be a variable (subscripted or not subscripted), or an expression.

EXAMPLES

```
A = COSF (B)
A = SQRTF (BETA)
Y = A — SINF (B* SQRTF (C) )
```

For the last example, the assembled instructions of the object program will:

1. Branch to the square root subroutine to compute the value of C.
2. Multiply the square root value of C (obtained in Step 1) by B.
3. Branch to the SINF subroutine to compute the sine of the value obtained from Step 2.
4. Subtract the value computed so far from the variable A.
5. Replace the present value of the variable Y with the value of the complete expression.

Only one value is produced by a given library function. The mode of a library subroutine is determined by its argument.

EXAMPLES

COS (A)	floating point
ABSF (I)	fixed point

The relocatable library subroutines supplied with the 1620 FORTRAN II System, with the exception of Absolute Value Function (ABSF), will not accept fixed point arguments.

Approximation Methods and Estimated Errors

Results of the library subroutines are truncated, and, in general, errors are no greater than one in the last digit of the mantissa. Approximation methods and estimated errors for functional subroutines are described in greater detail in the following paragraphs.

1. *Logarithm.* The natural logarithm of the fractional part of the positive argument is evaluated by using a power series expansion. The exponent of the argument is multiplied by ln 10. The product is added to the logarithm of the fraction, and the sum is the logarithm of the argument. For an argument with its value A in the range $.99 < A \leq 1.01$, the leading digits of its logarithm will be zeros, and the result will contain less than f significant digits because of normalization. The maximum truncation error in the result is $\pm 10^{-f}$.
2. *Exponential.* The value of e^A, where A is the value of the argument, is calculated by using a series approximation for 10^A. For $|A| = 227.955924206\ldots$ an exponent overflow will result for $A > 0$ or exponent underflow for $A < 0$. The value of A is multiplied by log e and the product separated into an integer and a fractional part. The integer becomes the exponent of the result and the fractional part is used to produce its mantissa by series approximation. If A is greater than zero, the maximum error in the result is $\pm 5 \times 10^{-f}$.
3. *Cosine-Sine.* The cosine and sine functions of an argument with value A in radians are computed by using a series approximation for cosine A with sine A = cosine $(\frac{\pi}{2} - A)$. The value A is reduced to within the range $-\frac{\pi}{2} \leq A \leq \frac{\pi}{2}$ For arguments with exponents less than 03, the magnitude of the maximum truncation error in the mantissa of the result does not exceed 10^{-f}. Accuracy in the mantissa of the result decreases as the size of the argument (exponent 03 or greater) increases.
4. *Arctangent.* The arctangent function of an argument with value A is evaluated by using a series approximation. The result is given in radians. The maximum trunction error in the mantissa of the result is $\pm 10^{f}$, except for results with an exponent less than or equal to -2. The maximum error for these results is ± 1 in the $(f + 1)$ decimal place.
5. *Square Root.* The square root is derived by the odd integer method. The result is accurate to 1 in the last digit of the mantissa.
6. *A ** B.* A^B is evaluated as EXPF (B*LOGF (A)). Three subroutines, logarithm, multiply, and exponential, are involved. An error in one of these subroutines may propagate other errors or increase the error in a succeeding subroutine. Normally, the magnitude of the error does not exceed 10^{1-f}.

Additional Library Functions

Up to 43 additional functions can be added to the library of subroutines. The user must code the new subroutine in 1620 SPS language with the origin defined at core location 10,000. The P and Q addresses of an instruction which are relative to address 10,000 must be indicated by placing flags over the O_0 and O_1 digits (the Op code digits) respectively, so that these addresses can be modified properly when the subroutine is relocated in storage.

When programming a subroutine with variable length floating point numbers, it may be desirable to use certain addresses and constants available in the arithmetic and input/output subroutines. A reference to the program listings of these subroutines will yield the information on these addresses and constants. As the mode of operation (fixed or floating point) is determined by the argument of the subroutine, the FORTRAN II compiler does not distinguish between fixed point and floating point subroutines. It is up to the user to have a thorough knowledge of the added subroutines and to use them correctly.

Linkage

In an object program the linkage to the library subroutines is in the form,

BTM	—SUBR, A	for a normal variable A, and
BT	—SUBR, A	if A is a parameter in a subprogram

where SUBR is the indirect address of the subroutine entrance, and A is the address of the argument. The A may or may not be in FAC.

It is imperative that space be provided at the end of each subroutine for storing the 5-digit address of the argument for the succeeding subroutine.

The total number of storage locations required must be even. Therefore, if the subroutine ends with a branch instruction (Op 49), then the instruction should be counted as 12 digits long in calculating the total length of the subroutine. If the subroutine ends with a Branch Back (Op 42), the instruction should be counted as at least 8 digits in length. In case the subroutine ends with a constant, then an additional 5 or 6 digits, depending on whether the constant has an even or odd address, should be added to the length of the subroutine.

For arithmetic and input/output subroutines, the actual addresses of the subroutines are used in the linkage.

Work Areas

In writing the subroutines, the programmer may first move the argument into one of the work areas such as FAC, BETA or SAVE. In arithmetic subroutines the exponent of a floating point result is usually stored in SAVE before being moved to FAC. A careful study of the arithmetic subroutines may reveal that the relocatable subroutine to be added can share the normalization, sign determination, overflow, underflow, and error typeout sections. The value calculated by the subroutine must be left in FAC. Even if no value is calculated, it is advisable to place a constant in FAC.

The procedures to add a subroutine to the library are described under ADDING LIBRARY SUBROUTINES.

Arithmetic Statement Functions

An arithmetic statement function is defined by a single arithmetic statement and applies only to the program in which it appears.

All arithmetic statements defining functions must precede the first executable statement of the program.

The name assigned to an arithmetic statement function must conform the the following rules.

1. The name must consist of at least one, but not more than six characters.
2. It must begin with an alphabetic character; numeric and alphabetic characters may follow the first character in any combination.
3. No special characters may be used.

The function name determines the mode of the value that is computed; for example, if the function name begins with I through N, the mode of the value will be fixed point.

The function statement is defined as follows.

NAME (ARG) = E

where NAME is the name of the function; ARG is the argument which consists of one or more non-subscripted variables, separated by commas and enclosed in parentheses; and EXP is an expression which conforms to the rules for forming expressions.

EXAMPLES

```
FIRST (X) = A * X + B
SECOND (X, B) = COS (X) * FIRST (B)
```

Any function name appearing in an expression must have previously been defined in a preceding statement.

The appearance of the name in an arithmetic statement serves to call the function. The value of the function (a single numeric quantity) is then computed, using the arguments which are supplied in the parentheses following the function name.

As many as desired of the variables appearing in the expression on the right-hand side may be stated on the left-hand side as arguments of the function. Since the arguments are really only dummy variables, their names are unimportant (except insofar as they indicate fixed-point or floating-point mode) and they may even be the same as names appearing elsewhere in the program.

Those variables on the right-hand side which are not stated as arguments are treated as parameters. Thus, if FIRST is defined in a function statement as FIRST (X) = A * X + B, then a later reference to FIRST (Y) will cause a y + b, based on the current values of a, b, and y, to be computed. The naming of parameters, therefore, must follow the normal rules of uniqueness.

The arguments of an arithmetic statement function reference may be expressions and may involve subscripted variables; thus, a reference to FIRST (Z + Y (I)), as a result of the previous definition of FIRST, will cause $a(z + y_i) + b$ to be computed on the basis of the current values of a, b, y_i and z.

Functions defined by arithmetic statements are always compiled as closed subroutines; that is, the machine-language instructions are compiled only once in the object program.

Dummy Variables Within an Arithmetic Statement Function

A variable appearing as a dummy argument within an arithmetic statement function must not have been defined previously except as a dummy argument in a previous arithmetic statement function. After the variable is used as a dummy argument, it may appear elsewhere in the program.

FORTRAN Functions

This class of functions covers subroutines that are not utilized frequently enough to be library functions, yet because of their size or complexity, they cannot be defined in a single arithmetic statement.

FORTRAN functions are compiled separately, yet the object program cannot be executed as an entity. For execution, they must be loaded (to core storage) and called by a main program. For this reason, FORTRAN functions are termed subprograms.

Subprograms allow large problems to be defined as a group of smaller problems, thus allowing several programmers to work concurrently in different areas of the same program. Since each subprogram is compiled separately, the arithmetic statement function name and variable names used in one subprogram are completely independent of the function and variable names used in the main program or other subprograms.

Subprograms are divided into two types: FUNCTION subprograms, and SUBROUTINE subprograms. Four statements, FUNCTION, SUBROUTINE, CALL, and RETURN are necessary for their definition and use. These statements are described later.

Although FUNCTION subprograms and SUBROUTINE subprograms are treated together and may be viewed as similar, they differ in two fundamental respects:

1. The FUNCTION subprogram can compute only one value, whereas the SUBROUTINE subprogram can compute many values (and then return them to the main program).
2. The FUNCTION subprogram is called by an arithmetic expression containing its name; the SUBROUTINE subprogram is called by the use of a CALL statement.

In all respects, subprograms must conform to the rules for FORTRAN II programming. They are usually compiled separately; however, they can be compiled together as described under BATCH COMPILATION.

FUNCTION Statement

'The FUNCTION statement, always first in a FUNCTION subprogram, defines it as a FORTRAN FUNCTION subprogram.

GENERAL FORM

FUNCTION Name $(a_1, a_2, \ldots, a_n)$

where Name is the symbolic name of a single-valued function, and each argument $a_1, a_2, \ldots, a_n$ (of which there must be at least one) is a nonsubscripted variable name. The function name consists of 1 to 6 alphabetic or numeric characters, the first of which must be alphabetic.

EXAMPLES

```
FUNCTION ARCSN(RADS)
FUNCTION ROOT (B, A, C)
FUNCTION INTRT (RATE, YEARS)
```

In a FUNCTION subprogram, the name of the function must appear either in an input statement list, or at least once as the variable on the left-hand side of an arithmetic statement. An example of the latter is:

```
FUNCTION NAME (A, B)
.
.
.
NAME = A + B
.
.
.
RETURN
```

The value of the function is returned to the calling program. The mode of a FUNCTION subprogram is determined by its name.

EXAMPLES

FUNCTION AMAST (A, K) Floating point
FUNCTION IAMAST (A, K) Fixed point

The arguments following the name in the FUNCTION statement may be considered as "dummy" variable names; that is, during object program execution, other actual arguments are substituted for them. Therefore, the arguments which follow the function reference in the calling program must agree in number, order, and mode with those in the FUNCTION statement in the subprogram. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names must appear in similar DIMENSION statements within its respective program. *None of the dummy variables may appear in EQUIVALENCE statements in the FUNCTION subprogram.*

SUBROUTINE Statement

GENERAL FORM

SUBROUTINE Name ($a_1, a_2, \ldots, a_n$)

where Name is the symbolic name of a subprogram, and each argument, $a_1, a_2, \ldots, a_n$, if any is specified, is a nonsubscripted variable name. The name of the subprogram consists of 1 to 6 alphabetic or numeric characters, the first of which must be alphabetic.

EXAMPLES

SUBROUTINE MATMP (A, N, M, B, L, C)
SUBROUTINE QDRT (B, A, C, ROOT 1, ROOT 2)

The SUBROUTINE statement, always first in a SUBROUTINE subprogram, defines it as a SUBROUTINE subprogram. A subprogram introduced by the SUBROUTINE statement must be a FORTRAN program and may contain any FORTRAN II statements except FUNCTION, or another SUBROUTINE statement.

A SUBROUTINE subprogram must be referred to by a CALL statement in the calling program. The CALL statement specifies the name of the subprogram and its arguments.

Unlike the FUNCTION subprogram which results in the calculation of only a single numeric value, the SUBROUTINE subprogram uses one or more of its arguments to return results. Therefore, the arguments so used must appear on the left side of an arithmetic statement in the subprogram (or alternately, in an input statement list within the subprogram).

The arguments of the SUBROUTINE statements are dummy names that are replaced, at the time of execution, by the actual arguments supplied in the CALL statement. There must, therefore, be correspondence in number, order, and mode, between the two sets of arguments. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names must appear in similar DIMENSION statements within its respective program.

For example, the subprogram headed by

SUBROUTINE MATMP (A, N, M, B, L, C)

could be called by the main program through the CALL statement

CALL MATMP (X, 5, 10, Y, 7, Z)

where dummy variables, A, B, C, are the names of matrices. A, B, C must appear in a DIMENSION statement in subprogram MATMP, and X, Y, Z must appear in a DIMENSION statement in the calling program. The dimensions assigned must be the same in both statements.

None of the dummy variables may appear in EQUIVALENCE statements in the SUBROUTINE subprograms. These subprograms may be independently compiled or used in a multiple compilation with others (see BATCH COMPILATION).

CALL Statement

The CALL statement refers only to the SUBROUTINE subprogram, whereas the RETURN statement (described later) is used by both the FUNCTION and SUBROUTINE subprograms.

GENERAL FORM

CALL Name ($a_1, a_2, \ldots, a_n$)

where Name is the name of a SUBROUTINE subprogram, and $a_1, a_2, \ldots, a_n$ are the arguments.

EXAMPLES

CALL MATMP (X, 5, 10, Y, 7, Z)
CALL QDRT (P*9.732, Q/4.536, R −S**2.0, X1, X2)

This statement is used to call SUBROUTINE subprograms; the CALL transfers control to the subprogram and presents it with the parenthesized arguments. Each argument must be one of the following types:

1. Fixed point constant.
2. Floating point constant.

3. Fixed point variable, with or without subscripts.
4. Floating point variable, with or without subscripts.
5. Arithmetic expression.

The arguments presented by the CALL statement must agree in number, order, mode, and array size with the corresponding arguments in the SUBROUTINE statement of the called subprogram, and none of the arguments may have the same name as the SUBROUTINE subprogram being called.

RETURN Statement

EXAMPLE

```
                    RETURN
```

This statement terminates any subprogram of the type headed by either a SUBROUTINE or a FUNCTION statement, and returns control to the calling program. A RETURN statement must, therefore, be the last executed statement of the subprogram. It need not be the last statement of the subprogram physically, but can be any point reached by a path of control. Any number of RETURN statements may be used.

Compilation Process

The process of compiling a FORTRAN II program is divided into two passes or machine runs. The first pass uses one deck (Pass I deck) of the compiler to process source statements and to produce an intermediate output. The second pass utilizes a second deck (Pass II deck) of the compiler and the intermediate output from Pass I to produce the object program and its loader.

Subroutines may be included in the object deck or loaded separately at object time. If any subprograms are to be included, they are loaded at object time when they are called for by the main program.

NOTE: If compilation is being done on a 1620 Model 2, the indexing feature must be in the "no band" mode.

Arithmetic and Input/Output Subroutines

The arithmetic and input/output subroutines, including constants and work storage, are basic routines needed for the proper execution of the object program. They are loaded without being specifically called for by the object program, and are stored in locations 00402 through 10999 at object time. Besides performing the fundamental tasks of adding, subtracting, etc., these routines also perform some diagnostic testing on the data being manipulated.

The arithmetic and input/output subroutines available with FORTRAN II are shown in Table 1. By referring to the symbolic names for the subroutines in the listing, their equivalent absolute addresses can be found.

Table 1. Arithmetic and Input/Output Subroutines

Subroutine	Symbolic Name	Operation
Floating Point Arithmetic		
Add	FAD	FAC + A → FAC
Subtract	FSB	FAC - A → FAC
Reverse Subtract	FSBR	A - FAC → FAC
Multiply	FMP	FAC x A → FAC
Divide	FDV	FAC / A → FAC
Reverse Divide	FDVR	A / FAC → FAC
Set FAC to zero	ZERFAC	0 → FAC
Fixed Point Arithmetic		
Add	FXA	FAC + I → FAC
Subtract	FXS	FAC - I → FAC
Reverse Subtract	FXSR	I - FAC → FAC
Multiply	FXM	FAC x I → FAC
Divide	FXD	FAC / I → FAC
Reverse Divide	FXDR	I / FAC → FAC
Common Subroutines		
Load FAC	TOFAC	A → FAC or I → FAC
Store FAC	FMFAC	FAC → A or FAC → I
Reverse Sign of FAC	RSGN	- FAC → FAC
Fix a Floating Point Number	FIX	FIX (FAC) → FAC
Float a Fixed Point Number	FLOAT	FLOAT (FAC) → FAC
Exponentiation		
Fixed Point J ** I	FIXI	FAC ** I → FAC
Floating Point A ** (±I)	FAXI	FAC ** (±I) → FAC
Floating Point A ** (±B)	FAXB	FAC ** (±B) → FAC
Input/Output		
Read Card	RACD	
Read Tape	RAPT	
Read Typewriter	RATY	
Write Card	WACD	
Write Tape	WAPT	
Write Typewriter	WATY	

FAC - simulated accumulator
A & B - floating point variables
I & J - fixed point variables
→ - store in

Pass I

Compiler Deck

The composition of the Pass I compiler deck is shown below.

Card Numbers (columns 76-80)	
00001-00009	Loader
00010-00413	Pass I compiler program
02000-02005	Multiply-Add tables
03000	Library subroutine count and names of library subroutines

Operating Procedures

The sequence of operations required to process a source program is as follows:

1. Set the Program switches for Pass I compilation (see Table 2).
2. Set the Overflow Check switch to PROGRAM and all other Check switches to STOP.
3. Press the Reset key.
4. Ready the card punch by loading blank cards into the punch hopper and pressing the Punch Start key.
5. Load Pass I of the compiler program by placing the deck in the read hopper and pressing the Load key. The cards are punched with sequence numbers in columns 76-80 and must be loaded in sequence. When the Pass I deck is successfully loaded, the console typewriter types out the following instruction,

   ENTER SOURCE PROGRAM, PRESS START

   and the 1620 halts.

Table 2. Program Switch Settings for Passes I and II

Switch	PASS I		PASS II	
	ON	OFF	ON	OFF
1	Source statements are listed as they are processed. Source statement errors are listed in the form ERROR n.* At end of pass, symbol table is listed out.	Source statements are not listed. Source statement errors are listed in the form SSSS + CCCC ERROR n. * Symbol table is not listed.	Statement numbers and addresses are listed. At the end of Pass II, subroutines are punched into object deck.	Statement numbers and addresses are not listed. At the end of Pass II, subroutines are not punched into object deck.
2	Enables batch compilation.	No batch compilation.	Not used.	Not used.
3	Source statements are entered at the console typewriter.	Source program is entered from card reader.	A trace instruction is compiled to trace the value of the expression generated in an IF statement. An additional instruction is generated in the object program for every IF statement.	Trace instructions for IF statements are not compiled.
4	Errors made while typing source statements can be corrected by a. turning on switch 4, b. pressing the Release and Start keys,	c. turning off switch 4 d. retyping statement.	Trace instructions for arithmetic statements are compiled but no additional instructions are generated.	Trace instructions for arithmetic statements are not compiled.

* See description under Errors in Source Program

6. Enter the source program statements through either the card reader or the console typewriter. If source statements are entered via the typewriter, each statement must be terminated by a record mark. A statement of up to 330 characters may be typed with no intervening punctuation, spacing, etc. Normally, card format need not be followed; however, in the case of a comment statement, the C must be followed by at least two blanks before the comment is typed. If the operator wishes to compile the program with *f* (floating point word length) and *k* (fixed point word length) not equal to 8 and 5, respectively, he should follow the procedure outlined under SOURCE PROGRAM CONTROL CARD.

Pass I Compiler Loading Errors

Sequence Errors. The Pass I deck of the compiler will not load if any of the first nine cards (loader) are out of sequence. If a sequence error is found, it must be corrected and the deck reloaded.

Following the loader, the cards numbered 00010 through 02000 are sequence-checked. If any of these cards are not in sequence, the message,

CARD OUT OF SEQUENCE

is typed. The card read out of sequence will be the second one from the back in the reader stacker. To restart,

1. Remove the cards remaining in the hopper.
2. Press the Nonprocess Runout key.
3. Remove the last two cards from the stacker.
4. Arrange the cards (gathered in 1, 2, 3 above, in proper sequence and press the Reader Start and Start keys.

Library Function Declaration Errors. After the main body of the Pass I deck is loaded, the program reads in the library function declaration card(s) at the end of the deck. If the message,

LIBRARY FUNCTION DECLARATION ERROR

is typed, it signifies that one of the following error conditions has been detected.

1. Library function declaration cards are out of sequence.
2. More than 50 library functions have been declared.
3. Library function name has more than 6 alphameric characters.
4. Library function name does not begin with an alphabetic character.

To restart after any one of these conditions occurs,

a. Remove the library function declaration cards from reader (first function card will be numbered 03000).
b. Correct the error.
c. Reload function cards.
d. Press the Reader Start and Start keys.

Entering the Source Program

The form of the source program can be either a punched deck or a list of statements to be typed in at the console typewriter. This entry option is exercised by positioning Program Switch 3 as follows:

1. Card input – Switch 3 off
2. Typewriter input – Switch 3 on

When typewriter input is selected, the compiler transfers control to the 1620 console to await typing of the first statement. Each typed statement must terminate with a record mark. After a statement is typed, the operator should press the Release and Start keys to process that statement. As soon as the statement is processed, the carriage returns to await entry of the next statement. After an END statement (the last statement of a source program) is processed and provided there have been no Type 1 errors (see Table 3), the message,

TURN SW 1 ON FOR SYMBOL TABLE, PRESS START

is typed on the console typewriter. If the operator does not want a symbol table listing, he should ascertain that Switch 1 is off, and press the Start key. The message,

END OF PASS I

will then be typed. If a symbol table listing is requested, this message is typed following the listing.

Page Heading Card

The source program may be preceded by an identification card containing the following information.

Columns 1-2	**
Columns 3-72	Identification data

If Switch 1 is on, the identification data will be typed out as a heading at the start of Pass I, at the start of the symbol table, and at the start of Pass II. This data will be typed out at the start of Pass I, regardless of the switch setting, if a source program control card is used.

The page heading card is punched out as the first card of the intermediate output and as the first card of the object deck. The first card of the intermediate output and the first card of the object program will be blank if the page heading card is omitted from the source program. Record marks and group marks should not be present in the card.

Source Program Control Card

The compiler will process an object program with a floating point mantissa length (f) of 8 digits and a fixed point word length (k) of 5 digits and assume the object machine has the same size core storage. The operator may vary these by using a control card. The control card must precede the source program, and follow the page heading card if one is present. The card should have the following format,

Column 1	* (asterisk)
Columns 2-3	ff
Columns 4-5	kk
Column 6	s
Columns 7-80	Not used

where ff is the floating point mantissa, kk is the fixed point word length, and s is the object machine core size ($s = 2$, 4, or 6 for 20,000, 40,000, or 60,000 positions).

If entry is from the typewriter, the following configuration should be typed preceding the source program.

*ffkks

If s is left blank, the compiler assumes the object and source machines to be the same size. The range of f is 2 through 28; of k, 4 through 10. If either f or k is out of the prescribed range, the message,

ERROR, F OR K OUTSIDE RANGE

is typed. To restart, replace the control card with a correct card and press the Reader Start key. If entry is from the typewriter, press START, type the correct values, and press the Release and Start key.

No correlation is necessary between f and k unless there is equivalence between fixed and floating point variables, in which case k must equal $f + 2$.

Errors in the Source Program

During compilation of Pass I, a number of tests are made for source program errors. If an error is found in a source statement and Switch 1 is on, a message in the form,

ERROR n

is printed, where n is the error code (see Table 3). If Switch 1 is off, the error message is in the form,

ssss + cccc ERROR n

where ssss is the last *statement number* encountered by the program prior to the error, and cccc −1 is a count of how many statements lie between the last numbered statement and the statement in error. For example,

509 +12

means the twelfth statement following the statement numbered 509. If an error occurs before a statement number is encountered, ssss will be blank. Comment

Table 3. Source Program Errors

TYPE 1: Compilation continues but punching of intermediate output is stopped. Only one error of this type is detected in any one statement.

Error No.	Condition
1	Undeterminable, misspelled, or incorrectly formed statement.
2	Syntax error in a nonarithmetic statement (exception: DO statements).
3	Dimensioned variable used improperly, i.e., without subscripting; or subscripting appears on a variable not previously dimensioned.
4	Symbol table full (processing may not be continued).
5	Incorrect subscript.
6	Same statement number assigned to more than one statement.
7	Control transferred to FORMAT statement.
8	Variable name greater than 6 alphameric characters.
9	Variable name used both as a nondimensioned variable name and as a Subroutine or Function name.
10	Invalid variable within an EQUIVALENCE statement.
11	Subroutine or Function name or dummy variable used in an EQUIVALENCE statement (subprogram only).
12	k not equal to f + 2 for equivalence of fixed point to floating point variables.
13	Within an Equivalence list, placement of two variables previously in Common, or one variable previously equivalenced and another either equivalenced or placed in Common.
14	Sense switch number missing in an IF (Sense Switch n) statement.
15	Statement number or numbers missing, not separated by commas, or nonnumeric in a transfer statement.
16	Index of a computed GO TO missing, invalid, or not preceded by a comma.
17	Fixed point number greater than k digits.
18	Invalid floating point number.
19	Incorrect subscripting within a DIMENSION statement.
20	First character of a name not alphabetic.

Error No.	Condition
21	Variable within a DIMENSION statement previously used as a nondimensioned variable, or previously dimensioned, or used as a Subroutine or Function name.
22	Dimensioned variable used within an arithmetic statement function.
23	More than four continuation cards.
24	Statement number in a DO statement appeared in a previous statement.
25	Syntax error in a DO statement.
26	FORMAT number missing in an input/output statement.
27	Statement number in an input/output statement appeared previously on a statement other than a FORMAT statement, or a number on a FORMAT statement appeared in other than an input/output statement.
28	Syntax error in input/output list, or an invalid list element.
29	Syntax error in CALL statement, or an invalid argument.
30	SUBROUTINE or FUNCTION statement not the first statement in a subprogram.
31	Syntax error or invalid parameter in a SUBROUTINE or FUNCTION statement.
32	Syntax error or invalid variable in a COMMON statement.
33	Variable in a Common list previously placed in Common or previously equivalenced.
34	Function name appeared to the left of an equal sign or input/output statement.
35	Syntax error in FORMAT statement, or invalid FORMAT specifications.
36	Invalid expression to the left of an equal sign in an arithmetic expression.
37	Arithmetic statement function preceded by the first executable statement.
38	Invalid expression in an IF or CALL statement, or invalid expression to the right of an equal sign in an arithmetic statement.
39	Unbalanced parenthesis.
40	Invalid argument used in calling an Arithmetic statement function or Function subprogram.

TYPE 2: Both compilation and the punching of intermediate output continue.

Error No.	Condition
51	DO loop ended with a transfer statement.
52	No statement number for next executable statement following a transfer statement.
53	Improperly ended nonarithmetic statement.
54	Unnumbered CONTINUE statement.
55	Number of Common addresses assigned in excess of storage capacity because of Equivalence. See note at end of Table.

Error No.	Condition
56	Statement number or subscript greater than 9999 (only first 4 significant digits are retained).
57	RETURN statement appeared in program other than a subprogram (statement ignored).
58	RETURN statement not contained in a Subroutine or Function subprogram.

NOTE: Error 55 is not detected if Type 1 errors occur during compilation.

cards, blank cards, and continuation cards are not included in the statement count. If a statement number is not defined, the message,

STATEMENT NUMBER SSSS UNDEFINED

is printed, where ssss is the undefined statement number.

Pass II

Compiler Deck

The composition of the Pass II compiler deck is as follows.

Card Numbers (columns 76-80)	
04001-04009	Loader
04010-04493	Pass II compiler program
05000-05005	Multiply-Add tables

Operating Procedures

1. Set the Console Program switches for Pass II (Table 2).
2. Set the Overflow Check switch to PROGRAM, and all other check switches to STOP.
3. Press the Reset key.
4. Ready the card punch by loading blank cards into the punch hopper and pressing the Punch Start key.
5. Load the Pass II deck of the compiler by placing the deck in the reader hopper and pressing the Load key. The deck should be in proper sequence; only cards 04010 through 05000, however, are sequence-checked by the compiler. If any of these cards is not in sequence, the error message,

   CARD OUT OF SEQUENCE

   is typed. For restart procedures, refer to PASS I COMPILER LOADING ERRORS. When the deck has been successfully loaded the computer will come to a programmed halt.
6. Place the intermediate output from Pass I into the reader hopper.
7. Press the Reader Start and Start keys.

Processing the Intermediate Output

The page heading card punched during Pass I must be the first card read by Pass II. As the intermediate output is being read, the card numbers (columns 77 through 80) are tested for proper sequence. In the mainline program, the number starts with 0001; in subprograms, 0000. If the cards are not in sequence, the message,

CARDS NOT IN ORDER

is typed and the machine halts.

To restart,

1. Remove the last two cards from the reader stacker.
2. Remove the cards that remain in the reader hopper.
3. Press the Nonprocess Runout key.
4. Arrange cards (from 1, 2, 3 above) in proper sequence, load them in the reader, and press the Reader Start and Start keys.

Overlap. During processing of the intermediate output, a test is made to determine whether the compiled object program (not including relocatable library subroutines), together with the required data, will occupy more core storage than is available. If this condition is found to exist, the message,

$\bar{x}$XXX OVERLAP

is typed immediately after the processing of the statement which caused the overlap. Here $\bar{x}$xxx is the relative number of the statement within the program not counting storage allocation statements, comments, or blank cards. For example, if $\bar{x}$xxx is $\bar{0}$050, then the overlap is caused by the fiftieth statement in the program.

If an overlap exists, the message,

OVERLAP $\bar{N}$NNNN

is typed at the end of the compilation; $\bar{N}$NNNN being the total number of core storage positions overlapped.

Subroutines

If a mainline program (not a subprogram) is being compiled, the message,

SW 1 ON TO PUNCH SUBROUTINES, PRESS START

will be typed after all intermediate output has been processed. If the subroutines are to be included in the object deck, the operator must

1. Turn on Program Switch 1.
2. Load the subroutine deck.
3. Press the Reader Start and Start keys.

If the subroutine deck is to be read in when the object program is executed, Switch 1 should be turned off and the Start key pressed. When processing is completed, the message,

END OF PASS II

is typed.

When the subroutines are reproduced for inclusion in the object deck, the following error messages may occur:

SUBROUTINE OVERLAP $\bar{\text{X}}$XXXX

If a subroutine being punched overlaps the available space, further punching of that subroutine and succeeding called-for subroutines ceases. However, a total count of the overlapped locations of all subroutines is stored and printed out with the message.

LIBRARY SUBROUTINE MISSING

If one or more subroutines is missing, the 1620 will halt to allow corrective action. Proceed as follows:

1. Remove the last two cards from the reader stacker.
2. Remove the deck from the reader hopper.
3. Run out the two cards in the reader.
4. Arrange the cards (from 1, 2, 3 above) in proper sequence.
5. Place the required relocatable subroutines ahead of the deck.
6. Place the deck back into the reader hopper and press the Reader Start and Start keys.

If the operator does not know which subroutine(s) is missing, he may print out storage locations 00415-00464 at the console typewriter. If the *n*th digit(s) is a 1 (one), the subroutine(s) numbered *n* was called but not loaded.

MISSING HEADER CARD

If the subroutine header card is missing, follow Steps 1, 2, 3, as outlined for LIBRARY SUBROUTINE MISSING, then

4. Arrange the cards in proper sequence and ascertain that the first card is a header card.
5. Place the deck back into the reader hopper and press the Reader Start and Start keys.

At the end of Pass II, the messages,

XXXXX LENGTH
YYYYY NEXT COMMON

are printed, where XXXXX is the number of core positions the object program requires (mainline program length includes length of Arithmetic and I/O subroutines), and YYYYY is the next available core storage position of the COMMON area(YYYYY + 1 is the last reserved position of COMMON).

Errors in the Source Program

During compilation of the intermediate output, some diagnostics are performed which were not performed during Pass I. If an error is detected, an error message in one of the following forms is printed.

XXXX SYMBOL TABLE FULL
XXXX IMPROPER DO NESTING
XXXX MIXED MODE
XXXX DO TABLE FULL

Here, XXXX is the relative number of the statement within the program, not counting storage allocation statements, comments, or blank cards. The number XXXX does not correlate with any actual statement number.

If an IMPROPER DO NESTING or MIXED MODE error is detected, compilation continues; however, the output will not be usable. If the message, SYMBOL TABLE FULL, is printed (due to too many symbols), compilation stops. The DO TABLE FULL message appears when more than 29 DO statements are nested; compilation does not continue.

Batch Compilation

Batch compilation is a method by which several inputs can be processed and the output obtained without reloading the compiler program.

1620 FORTRAN II allows batch compiling for both Pass I and Pass II.

Pass I

To batch-compile for Pass I, proceed as follows:

1. Remove intermediate output from the punch hopper after the message END OF PASS I has been typed.
2. Turn on Program Switch 2. (Program Switch 2 may be turned on at any time during Pass I.)
3. Press the Punch Start key on the card punch and press the 1620 Start key. This will cause a symbol table deck containing a 3-digit sequence number in columns 78 through 80 (numbering starts with 101) to be punched out.
4. After the necessary initialization is done, the following message is typed:

   ENTER SOURCE PROGRAM, PRESS START

   Set the program switches as desired and proceed to enter the source statements.

Pass II

To batch-compile for Pass II, proceed as follows:

1. Place deck containing symbol table between the page heading card and the remainder of the intermediate output deck from Pass I, and load both into the card reader.

2. Press the Reader Start key.
3. After the output is obtained, load another intermediate output with its page heading card and symbol table and press the Reader Start key.

If the operator wishes, he may load a series of decks on top of each other, taking care to keep respective symbol tables between the page heading card and the intermediate decks. The output will be one composite deck which will have to be separated manually.

Object Program

Description

A 1620 FORTRAN II object program will most likely consist of two parts: a mainline program and a group of subprograms. Although these two segments are similar in most respects, they will be described separately to show their relative position in, and importance to, the over-all objective.

Discussion of subroutines in this section will be limited to the way they affect the loading of the object program.

Mainline Program

As the name implies, a mainline program exercises control over the entire object program. In addition to the normal execution of instructions, it has the ability to call subprograms and subroutines when they are needed.

Makeup. The mainline program is comprised of three segments:

1. Loader (cards $\bar{0}00001$ - $\bar{0}00035$). Located at the beginning of the object deck, this loader is used for loading both the mainline program and any subprograms that are called.
2. Communication card (card $\bar{0}00036$).

   This card contains:

Columns 1-2	Value of f.
Columns 3-4	Value of k.
Columns 5-9	Address of first instruction in the object program.
Columns 10-14	Next available address in the Common area.

   It is loaded into locations 00401 - 00414.
3. Object program (card $\bar{0}00037$ to end of deck). FORTRAN II employs five different types of cards in the mainline object deck. The cards are identified by the character punched in column 62, as follows:

   0 *Instructions.* One to five relocatable instructions are contained in columns 1-60 of an instruction card. The P or Q fields of the instructions are modified for relocation depending upon whether a flag is present over the O_0 (for P) or O_1 (for Q) position of the operation code. The loading addresses are punched in columns 65-69 and 70-74.

   1 *Constant.* Only the loading addresses are modified.

   $\bar{0}$ *Relative addresses.* One to twelve relocatable 5-digit addresses are contained in columns 1-60 of this card. As in a constant card, the loading addresses of this card are also modified.

   $\bar{1}$ *Nonrelocatable constants.* This card contains constants which are not relocatable; hence the loading address are not modified.

   ‡ *Subprogram calling card.* The name in numeric representation of the subprogram being called is right-justified in columns 1-12. The address where the starting address of the subprogram (after relocation) is to be placed is contained in columns 13-17.

The loading addresses in each card are as follows.

Columns 65-69	leftmost address where instructions or constants will be loaded.
Columns 70-74	rightmost address plus one, where instructions or constants will be loaded.

A card containing a 1 in column 63 indicates to the loader the end of the mainline program or subprogram.

Subprograms

Subprograms are compiled in the same manner as mainline programs, except they do not have their own loader. Therefore they can be loaded only when called for by a mainline program or another subprogram.

Makeup. A subprogram is made up of a header card and a number of program cards. The header card contains:

Columns 1-12	Subprogram name, right-justified, in numeric representation.
Columns 21-22	Value of f (floating point mantissa length).
Columns 23-24	Value of k (fixed point word length).
Column 63	Record mark (identifies header card).

The subprogram cards have the same format as described for mainline program cards 37 and up.

Library Subroutines

If the subroutine deck was processed at compilation time, then the subroutine loader, any relocatable subroutines used by the object program, and the arithmetic and input/output subroutines are already a part of the object deck. If not, they will have to be loaded after the mainline program and the subprograms are loaded (see LOADING LIBRARY SUBROUTINES).

Makeup. The subroutine deck is composed of cards containing the following information.

For users without the Automatic Floating Point Operations feature:

Card Numbers (columns 75-80)	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010017	LOGF	848
020001-020025	EXPF	1132
030001-030019	COSF-SINF	882
050001-050026	ATANF	1256
060001-060011	SQRTF	536
070001-070004	ABSF	82
510001-510228	Arithmetic and Input/Output subroutines	11000
510229-510255	*Modifier cards	None
510256	Trailer card	None

For users with the Automatic Floating Point Operations feature:

Card Numbers	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010016	LOGF	824
020001-020024	EXPF	1108
030001-030018	COSF-SINF	846
050001-050026	ATANF	1232
060001-060011	SQRTF	536
070001-070003	ABSF	58
510001-510212	Arithmetic and Input/Output subroutines	11000
510213-510240	*Modifier cards	None
510241	Trailer card	None

Execution of the Object Program

Loading the Mainline Program

To load the mainline program,

1. Remove the first card of the object deck. (This is the page heading card.)
2. Place the mainline object deck into the card reader.

*The modifier cards alter the subroutines to accommodate the word length being used in the object program.

3. Set the Overflow switch to PROGRAM.
4. Set the Check switches to STOP.
5. Press the Reset key.
6. Press the Load key on the card reader.

Loading Subprograms

Subprograms are normally loaded directly back of the mainline program. The first card of the subprogram object deck is the page heading card which must be removed before the subprogram is loaded. If subprograms are called by the mainline program and have not yet been loaded, a printout will remind the operator to load subprograms (see ERROR MESSAGES for Loading Object Programs).

NOTE: If library subroutines were included in the object deck at compilation time, they must be separated from the mainline program before the subprograms are loaded. The procedure is:

1. Search object deck for beginning of library subroutines (card number 009970).
2. Place subprograms between the mainline program and the subroutines.
3. Place the deck into the reader hopper.
4. Press Load key on the card reader.

Loading Library Subroutines

After all subprograms are loaded, the typewriter types out the message,

LOAD SUBROUTINES

and the 1620 halts. If subroutines were processed at compilation time, the operator need only press the Start key to continue. However, if the subroutines were not previously processed, he must:

1. Place the subroutine deck into the card reader.
2. Press the Load key on the reader, or press the Reader Start and Start keys.

When the subroutines have been processed, the message,

ENTER DATA

is typed, and the machine halts.

After the data is entered, the operator presses the Start key to initiate execution of the object program.

Entering Input Data from the Typewriter

With each execution of an ACCEPT statement, the typewriter carriage returns, signaling the operator to type the input quantities corresponding to the variables in the ACCEPT statement list. Should the operator commit a typing error during console entry of data, he may recover by:

1. Turning on Program Switch 4,
2. Pressing the Release and Start keys,
3. Turning off Program Switch 4, and
4. Re-entering the complete record of data.

This process may be repeated as often as necessary.

Trace Feature

Under program switch control, instructions are compiled into the object program, enabling the operator to trace the flow of the program for checking purposes. Program Switch 4 controls the trace feature and operates as follows:

	ON	OFF
Switch 4	Compiled trace instructions are executed.	Trace instructions are not executed.

The trace output provided contains the value of the left-hand side of each executed *arithmetic* statement, and the value of the expression calculated in an IF statement (if trace instructions for IF statements were compiled). Normal output resulting from PUNCH, PUNCH TAPE, PRINT, and TYPE statements is not inhibited. The output format of the trace data, printed at the left margin, is the same as that carried internally in the object program.

Note that Program Switch 4 serves a dual control function during execution of the object program: (1) the provision of trace data (2) the correction of input data erroneously entered at the console keyboard. Thus, when the machine is in the trace mode, the operator should turn off Switch 4 before typing input data. Following the last entry of the list, the operator should:

1. Press the Release key.
2. Press the SIE key two or three times.
3. Turn on Switch 4.
4. Press the Start key.

Care should also be exercised when using IF (Sense Switch 4) statements in a source program.

Error Messages

Loading Object Programs

1. Except for the first six cards in the object program deck, the card sequence number punched in columns 75-80 is checked during loading. If a card is out of sequence, the message,

   CARD OUT OF SEQUENCE

   is typed on the console typewriter and the loading process is halted.

   To resume loading:
   a. Remove the last two cards from the reader stacker.
   b. Remove the deck from the reader hopper.
   c. Run out the two cards in the reader.
   d. Arrange the cards in proper sequence (from a, b, c above).
   e. Place the deck back into the reader hopper.
   f. Press the Reader Start and Start keys.
2. A table of 100 entries is provided for storing information concerning the calling of subprograms. If more than 100 subprograms are called during the loading of the main object program, the message,

   SUBPROGRAM TABLE FULL

   is typed and loading is halted. The rest of the program cannot be loaded until two constants in the loading program are modified. After the constants are modified, the entire object program should be reloaded.

   The constants are located in columns 56-58 of Card 000007 and columns 8-10 of Card 000015. Both have the value 101 for 100 table entries and can be changed to any number not greater than 400. This limit is imposed for the following reasons:
   a. The table area starts at location 04001.
   b. Each table entry occupies 20 digits of storage.
   c. The main object program, including data and constants, starts at location 11000.
3. Each subprogram has a header card as the first card, and a trailer card as the last card of the object deck. If another header card appears between these two cards, the message:

   EXTRA HEADER CARD

   is printed and loading is halted.

   To resume loading, follow the same procedures as described for CARD OUT OF SEQUENCE with one exception; i.e., discard the second card (record mark, 0-2-8, in column 63) that was removed from the stacker. As in the previous procedure it is essential that the remaining partial deck be loaded in sequence.
4. The second card in a subprogram object deck must be an instruction card punched with a 0 (zero) in column 62. The message,

   INSTRUCTION CARD MISSING

   appears only when this card has the correct sequence number (000002) but does not have a zero punched in column 62.

Reloading is possible by following the same procedures outlined for CARD OUT OF SEQUENCE and ascertaining that the card with the number 000002 is an instruction card.

5. The header card of a subprogram deck contains the lengths of floating point mantissa (f) and fixed point integers (k) in columns 21-22 and 23-24, respectively. If the subprogram is called, these lengths are compared with the corresponding f and k values in the main program. Both values must agree or the message,

   SUBPROGRAM HAS DIFFERENT F OR K

   is printed and loading stops.

   To resume loading, follow the procedures below:
   a. Remove the last two cards from the reader stacker.
   b. Remove the deck from the reader hopper.
   c. Press the Nonprocess Runout key.
   d. Remove the deck that did not compare and replace it with the correct deck.
   e. Place the entire deck back into the reader hopper.
   f. Press the Reader Start and Start keys.
6. After the object program is completely loaded, the names of subprograms called but not loaded are typed, followed by the message,

   READ SUBPROGRAMS NAMED ABOVE

   Since it is possible for one subprogram to call another, the deck of subprograms may be so arranged that several loading passes are necessary to place all called subprograms into core storage. For example, if the main program calls for subprogram A and this subprogram in turn calls for subprogram B, and B precedes A in the deck, then subprogram B is bypassed the first time. After loading subprogram A, subprogram B must be reloaded. The programmer can eliminate these multiple passes by properly arranging the deck of subprograms.

   When the message is printed, the operator must:
   a. Press the Nonprocess Runout key.
   b. Place the required subprograms(s) into the hopper.
   c. Press the Reader Start and Start keys.
7. If a non-blank card is read after the trailer card of an object program (mainline or subprogram), the message,

   EXTRA CARD AT END OF DECK

   is typed and the 1620 halts.

   Remove the extra card from the reader stacker and run out the cards in the reader. Examine the deck to assure its correctness, and replace it into the hopper. Resume loading by pressing the Reader Start and Start keys.

Loading Library Subroutines

1. During loading, the sequence numbers of all relocatable subroutines are checked; the subroutine loader is not sequence-checked. If a card is out of sequence, the message,

   CARD OUT OF SEQUENCE

   is typed and the loading process is halted.

   To resume loading:
   a. Remove the last two cards from the reader stacker.
   b. Remove the deck from the reader hopper.
   c. Run out the two cards in the reader.
   d. Arrange the cards (from a, b, c above) in proper sequence.
   e. Place the deck back into the reader hopper.
   f. Press the Reader Start and Start keys.

   The cards for the arithmetic and input/output subroutines are not checked for proper sequence because of storage limitations.
2. The message,

   SUBROUTINE HEADER MISSING

   appears if the twenty-third card in the library subroutine deck is not a header card.

   To resume loading:
   a. Remove the last two cards from the reader stacker.
   b. Remove the deck from the reader hopper.
   c. Run out the two cards in the reader.
   d. Arrange the cards (from a, b, c above) in proper sequence and ascertain that the first card is a header card (identified by a 0-2-8 punch in column 63).
   e. Place the deck back into the reader hopper.
   f. Press the Reader Start and Start keys.
3. When all the selected relocatable subroutines in the deck have been loaded, a check is made to determine if any required subroutines have not been loaded. If one or more of these subroutines is missing, the message,

   LIBRARY SUBROUTINE MISSING

   is printed, and the loading process is halted.

   The operator can determine which subroutine(s) is missing by comparing his program requirements with the subroutine deck. An easier method is to print out storage locations 00415-00464 (fifty digits) by inserting:

```
38 00415 00100
48 00000 00000
41
```

at the console typewriter. If the *n*th digit(s) is a 1 (one), then the subroutine(s) numbered *n* was called but not loaded.

Loading can be resumed by following these procedures:

a. Remove the last two cards from the reader stacker.
b. Remove the deck from the reader hopper.
c. Run out the two cards in the reader.
d. Arrange these cards in proper sequence. The first card should have 5̄10001 punched in columns 75-80.
e. Place the required subroutines ahead of the deck.
f. Place the deck back into the reader hopper.
g. Press the Reader Start and Start keys.

4. If the capacity of core storage is exceeded during the loading of subprograms and library subroutines, the overlapping data will not be loaded into storage. At the end of reading the library subroutines, the message,

   OVERLAP x̄xxxx DIGITS

   is typed and the 1620 halts *before* loading the rest of the subroutine deck. (If the subprograms caused the overlap, the message is not typed until this time.) In the error message, x̄xxxx is the total number of digits overlapped.

   Since it is not possible to execute the object program without the required subprograms and subroutines, the operator should discontinue loading and then revise his source program.

Subroutine Error Checks

A number of error checks have been built into the library subroutines. The basic philosophy in the disposition of an error is to print an error message, set the result of the operation to the most reasonable value under the circumstances, and continue with the program. Subroutine error codes, the nature of the error, and the value of the result in FAC (symbolic name of the accumulator in which arithmetic operations are performed) are listed in Table 4.

In the Table, the terms "overflow" and "underflow" appear several times. Overflow means that the exponent of the result has exceeded 99; underflow means that the exponent of the result is less than —99. The result of overflow is affixed with the proper sign (not indicated in Table 4).

Table 4. Subroutine Error Codes

Error Code	Error	Result in FAC
E1	Overflow in FAD or FSB	9̄9......99̄9
E2	Underflow in FAD or FSB	0̄0......09̄9̄
E3	Overflow in FMP	9̄9......99̄9
E4	Underflow in FMP	0̄0......09̄9̄
E5	Overflow in FDV or FDVR	9̄9......99̄9
E6	Underflow in FDV or FDVR	0̄0......09̄9̄
E7	Zero division in FDV or FDVR	9̄9......99̄9
E8	Zero division in FXD or FXDR	9̄99.....
E9	Overflow in FIX	9̄9......
F1	Loss of all significance in FSIN or FCOS	0̄00.....09̄9̄
F2	Zero argument in FLN	9̄9......9̄9̄9
F3	Negative argument in FLN	ln /x/
F4	Overflow in FEXP	9̄9......99̄9
F5	Underflow in FEXP	0̄0......09̄9̄
F6	Negative argument in FAXB Negative argument in FSQR	$/A/^B$ SQR /x/
F7	Input data in incorrect form or outside the allowable range	
F8	Output data outside the allowable range	
F9	Input or output record longer than 80 or 87 characters (whichever is applicable to the I/O medium being used)	
G1	Zero to minus power in FIXI	9̄99.....
G2	Fixed point number to negative power in FIXI	0̄00.....
G3	Overflow in FIXI	9̄99.....
G4	Floating point zero to negative power in FAXI	9̄9......99̄9
G5	Overflow in FAXI	9̄9......99̄9
G6	Underflow in FAXI	0̄0......09̄9̄
G7	Zero to negative power in FAXB	9̄9......99̄9

The error printout is in the form:

ER XX

where xx is the error code in the table.

If Error F7 occurs during the execution of the instructions compiled for an input statement, the field which is incorrect is replaced by zeros, and processing continues.

The exponent portion of an E-type input data field must be right-justified in that field and may contain only one sign. Deviations from this rule are not checked. For exponents greater than 99 (absolute value), the value is reduced modulo 100.

If Error F8 occurs, the incorrect field is set to blanks in the output record, and an additional record is printed. This record contains the incorrect field in the form:

E $(f+6)$.f	for floating point numbers, and
I $(k+1)$	for fixed point numbers.

This additional record is also produced on the output device (card punch, tape punch, or typewriter) called for by the source statement.

If Error F9 occurs, the incorrect field is ignored and processing continues. However, it is possible that part of the subroutines or object program may have been destroyed by the abnormal record. In this case, the program may inexplicably halt at some point during its execution.

Adding Library Subroutines

Any library subroutines, with the exception of logarithm and exponential (needed for evaluating A**B), may be deleted or replaced with new subroutines. Up to 50 subroutines can be accommodated by the FORTRAN II System.

Addition of a library subroutine involves the modification of the compiler deck (Pass I) and the library subroutine deck.

Modifying the Compiler Deck

At the end of the Pass I deck, a card (numbered $\bar{0}3000$) is used to indicate the total number of relocatable subroutines and their names. This card has the total number of library subroutines punched in columns 1 and 2; for example, $\bar{0}7$ for the seven subroutines provided with the system. If no subroutines were included with the system, then 00 would be punched in columns 1 and 2.

This card also contains the names and synonyms of the subroutines. The names may not be composed of more than six characters. The synonyms are separated by commas with no intervening spaces. The set of synonyms for each subroutine is terminated by a period.

Adding a subroutine involves revision of the total number of subroutines punched in card $\bar{0}3000$, and an addition to the name card(s). Assuming a hyperbolic cosine subroutine is to be included in the library subroutines as the eighth relocatable subroutine, the operator should:

1. Change the number in columns 1-2 of card $\bar{0}3000$ to $\bar{0}8$.
2. Punch the symbol name(s), for example COSH, COSHF in the columns following the last subroutine name (ABSF). If the set of names and synonyms is too long to fit in the remaining columns of card $\bar{0}3000$, start the *entire* set in a new card with a sequence number of $\bar{0}3001$ in columns 76-80.

Since the subroutine names are identified internally by their sequential position in the name card(s), the names and cards must be in strict sequence. Care must also be exercised when deleting subroutines so a gap will not be created.

Modifying the Library Subroutine Deck

The new subroutine must be assembled by means of the 1620 SPS compiler. The resulting condensed object deck, with the first two and the last seven cards removed, must be preceded by a header card punched with the following information:

Columns 1-5	ZZZZZ	Total number of storage locations required by the subroutine. This number must be *even.*
Columns 11-12	YY	The alternate subroutine number, if any.
Columns 16-20	WWWWW	The alternate entry point, if any.
Column 63	‡	A record mark (0-2-8)
Columns 75-80	$\bar{X}$X0001	Card sequence number, where XX is the subroutine number.

The object deck, minus the header card, must be renumbered starting with $\bar{X}$X0002, where XX is the subroutine number. This number, XX, must agree with the number punched in columns 79-80 of the name card in the Pass I deck.

If a subroutine contains two entry points, for example, cosine at address 10,000 and sine at address 10,036, its deck must be numbered with XX as the subroutine whose entrance is at 10,000. The other subroutine number, YY, becomes the alternate and is punched in columns 11-12. If the cosine subroutine is assigned the number 03 and the sine subroutine, number 04, the deck is numbered starting at $\bar{0}3001$ with 04 punched in columns 11-12 to indicate the alternate number. Columns 16-20 contain the alternate entrance WWWWW, which in this case is 00036 (10036 minus 10000).

The complete subroutine deck should be checked to see that all cards are sequentially numbered in columns 75-80. The packet of cards can now be inserted between any two subroutines in the library subroutine deck. Note that each subroutine is identified by the number in columns 75-76, and that all arithmetic and input/output subroutines are numbered 51.

Subroutine Versions

Two sets of library subroutines are available to the user; one contains subroutines for use on a 1620 System without the floating point feature; the other contains subroutines that may be used only on a 1620 System with the floating point feature installed. The first set mentioned may be used on a system that has the floating point feature installed; however, in that instance it is advantageous to use the second set of subroutines since execution time of the object program is reduced.

When the second set is used, the Pass II compiler deck must be modified by replacing cards numbered 04288 through 04292 with the cards numbered 04288-04292 which are included with Deck 4.

Modification of Origin

It is possible to change the origin of the object program when either of the two subroutine sets is used. More positions of core storage can be saved if the Automatic Floating Point Operations feature is installed; however, the following origin modifications apply to a 1620 System with or without the Automatic Floating Point Operations feature.

NOTE: FHO indicates that a flag is required in the high-order position of the location specified. ORIGIN is the lowest-numbered core storage location that the programmer desires to use.

	STANDARD ORIGIN	MODIFIED ORIGIN
Deck 1		
(Compiler, Pass I)		
Card No. 00269,		
Columns 54-58	11000(FHO)	ORIGIN+0(FHO)
Deck 2		
(Compiler, Pass II)		
Card No. 04474,		
Columns 40-44	11000(FHO)	ORIGIN+0(FHO)
Deck 3		
(Library subroutines)		
Card No. 09973,		
Columns 44-48	11000(FHO)	ORIGIN+0(FHO)
Card No. 510250,		
Columns 8-12	11001	ORIGIN+1
Columns 20-24	11005	ORIGIN+5
Columns 32-36	11006	ORIGIN+6
Columns 44-48	11010	ORIGIN+10
Deck 4		
(Library subroutines)		
(For use with the Automatic Floating Point Operations feature)		
Card No. 009973,		
Columns 44-48	11000(FHO)	ORIGIN+0(FHO)
Card No. 510238,		
Columns 8-12	11001	ORIGIN+1
Columns 20-24	11005	ORIGIN+5
Columns 32-36	11006	ORIGIN+6
Columns 44-48	11010	ORIGIN+10

Summary

To summarize, the procedures for adding a relocatable subroutine are as follows:

1. Increase the number in columns 1-2 of card 03000 of the Pass I deck to reflect a new total number of relocatable subroutines.
2. Add the subroutine name(s) at the end of the last card. If a new card is required, it must be numbered consecutively.
3. Write the subroutine in 1620 SPS language, observing the following rules:
   a. The origin of the subroutine is at address 10,000 (DORG 10000).
   b. The subroutine entrance is location 10,000.
   c. The address of the argument relative to 10,000 is made available at 9995-9999.
   d. The P and Q addresses in an instruction relative to 10,000 must be identified by flagging the O_0 and O_1 digits, respectively, in the instruction.
   e. The result must be left in FAC (or a constant placed in FAC if no result has been calculated).
4. Assemble and condense the subroutine. Remove the first two and last seven cards. Renumber the deck, starting at XX0002, where XX is the subroutine number.
5. Prepare a subroutine header card. Remember that the number of storage locations specified on the card must include an extra 5 locations for use by the next subroutine at object time.
6. Place the header card ahead of the subroutine cards and check their sequence numbers.
7. Place the packet of cards between any two subroutines in the library subroutine deck.

IBM 1620-1443 FORTRAN II Programming System

120 Print Position Systems

The 1620-1443 FORTRAN II Programming System is a printer-oriented version of the basic 1620 FORTRAN II System described in the preceding pages of this publication. It makes use of the IBM 1443 Printer as an integral unit during the processing of FORTRAN II source programs. The advantage of the printer-oriented system is that it provides a convenient and relatively fast means of listing source and object program information and the symbol table. Only the differences between the standard system and the printer-oriented system are described here. Specifications and operating procedures pertaining to the standard system are valid for the printer-oriented system if no specific mention is made of them in this section.

Language

The language specifications of the 1620-1443 FORTRAN II System are identical to the specifications of the 1620 FORTRAN II System with the exception of the PRINT and FORMAT statements. The specifications of these statements have been modified for use with the 1443 Printer as described in the following paragraphs.

PRINT Statement

When used with the 1620-1443 FORTRAN II System, the PRINT statement is used to print data on the 1443 Printer. The format of the PRINT statement is as follows:

PRINTn, List

where *n* is the statement number of a FORMAT statement, and *List* is a list of the quantities to be printed.

The TYPE statement is still used to type messages on the console typewriter.

FORMAT Statement

The FORMAT statement, when used with a PRINT statement, allows for a maximum of 120 characters for each printed line. (When used with a TYPE statement, the FORMAT statement is still limited to 87 characters.)

Carriage Control

In addition to the normal functions of a FORMAT statement, there is another function that it must perform when used with a PRINT statement. This function consists of designating a desired line-space or carriage-skip operation. A FORMAT statement, to be used in conjunction with a PRINT statement, will usually begin with 1H followed by a control character which specifies the desired operation. The control characters and their effects are:

blank	– Single space before printing
0	– Double space before printing
1-9	– Immediate skip to channels 1-9

The control character itself does not become part of the printed output.

EXAMPLE

```
      PRINT 2 A, B, J
2     FORMAT (1H0, F8.2, F8.2, I8)
```

This specification will provide for a double line-space between the line being printed and the *previous printed line.*

NOTE: The first carriage control specification is applicable only to the first line printed. If more than one line is called for, the carriage control specification must precede the normal specification for each line of print. For example:

```
FORMAT (1H0, F8.2, F8.2/1H0, F8.2, F8.2)
```

will provide for double line spacing before each printed line.

Compiler

Listings and Symbol Table Output

Listings and symbol table output appear on the printer instead of the console typewriter. The console switch settings used to obtain these outputs remain the same as those used in the standard system. The formats of the outputs have been changed to take advantage of the characteristics of the printer.

Error Messages

All error messages appear on the printer. Instructions to the operator still appear on the console typewriter.

Subroutine error code F9 has been modified to include any printer records that exceed 120 characters.

Trace Routine

If the trace routine is used, its output will appear on the printer.

Compiler Decks

Pass I

Card Numbers (columns 76-80)

00001-00009	Loader
00010-00410	Pass I compiler program
02000-02005	Multiply-Add tables
03000	Library subroutine count and names of library subroutines

Pass II

Card Numbers

04001-04009	Loader
04010-04491	Pass II compiler program
05000-05005	Multiply-Add tables

Subroutine Deck

The subroutine deck is composed of cards containing the following information.

For users without the Automatic Floating Point Operations feature:

Card Numbers (columns 75-80)	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010017	LOGF	848
020001-020025	EXPF	1132
030001-030019	COSF-SINF	882
050001-050026	ATANF	1256
060001-060011	SQRTF	536
070001-070004	ABSF	82
510001-510223	Arithmetic and Input/Output subroutines	11000
510224-510250	*Modifier cards	None
510251	Trailer card	None

*The modifier cards alter the subroutines to accommodate the word length being used in the object program.

For users with the Automatic Floating Point Operations feature:

Card Numbers	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010016	LOGF	824
020001-020024	EXPF	1108
030001-030018	COSF-SINF	846
050001-050026	ATANF	1232
060001-060011	SQRTF	536
070001-070003	ABSF	58
510001-510209	Arithmetic and Input/Output subroutines	11000
510210-510237	*Modifier cards	None
510238	Trailer card	None

Modification of Origin

When it is desired to change the origins of object programs, the following modifications must be made:

	STANDARD ORIGIN	MODIFIED ORIGIN
Deck 1		
(Compiler, Pass I)		
Card No. 00266,		
Columns 12-16	11000(FHO)	ORIGIN+0(FHO)
Deck 2		
(Compiler, Pass II)		
Card No. 04472,		
Columns 35-39	11000(FHO)	ORIGIN+0(FHO)
Deck 3		
(Library subroutines)		
Card No. 09973,		
Columns 44-48	11000(FHO)	ORIGIN+0(FHO)
Card No. 510250,		
Columns 8-12	11001	ORIGIN+1
Columns 20-24	11005	ORIGIN+5
Columns 32-36	11006	ORIGIN+6
Columns 44-48	11010	ORIGIN+10
Deck 4		
(Library subroutines)		
(For use with the Automatic Floating Point Operations feature)		
Card No. 009973,		
Columns 44-48	11000(FHO)	ORIGIN+0(FHO)
Card No. 510235,		
Columns 8-12	11001	ORIGIN+1
Columns 20-24	11005	ORIGIN+5
Columns 32-36	11006	ORIGIN+6
Columns 44-48	11010	ORIGIN+10

NOTE: FHO indicates that a flag is required in the high-order position of the location specified. ORIGIN is the lowest-numbered core storage location that the programmer desires to use.

144 Print Position Systems

The following differences should be noted when using the 1620-1443 FORTRAN II System with 144 positions.

FORMAT Statement

The FORMAT statement, when used with a PRINT statement, allows for a maximum of 144 characters per line.

Error Messages

All error messages appear on the printer. Instructions to the operator still appear on the console typewriter.

Subroutine error code F9 has been modified to include any printer records that exceed 144 characters.

Trace Routine

If the trace routine is used, its output will appear on the printer in E-type or I-type format.

Compiler Decks

Pass I

Card Numbers (columns 76-80)	
00001-00009	Loader
00010-00416	Pass I compiler program
02000-02005	Multiply-Add tables
03000	Library subroutine count and names of library subroutines

Pass II

Card Numbers	
04001-04009	Loader
04010-04490	Pass II compiler program
05000-05005	Multiply-Add tables

Subroutine Deck

The subroutine deck is composed of cards containing the following information.

For users without the Automatic Floating Point Operations feature:

Card Numbers (columns 75-80)	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010017	LOGF	848
020001-020025	EXPF	1132
030001-030019	COSF-SINF	882
050001-050026	ATANF	1256
060001-060011	SQRTF	536
070001-070004	ABSF	82
510001-510236	Arithmetic and Input/Output subroutines	11200
510237-510266	*Modifier cards	None
510267	Trailer card	None

*The modifier cards alter the subroutines to accommodate the word length being used in the object program.

For users with the Automatic Floating Point Operations feature:

Card Numbers	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010016	LOGF	824
020001-020024	EXPF	1108
030001-030018	COSF-SINF	846
050001-050026	ATANF	1232
060001-060011	SQRTF	536
070001-070003	ABSF	58
510001-510223	Arithmetic and Input/Output subroutines	11200
510224-510254	*Modifier cards	None
510255	Trailer card	None

Modification of Origin

When it is desired to change the origins of object programs, the following modifications must be made:

	STANDARD ORIGIN	MODIFIED ORIGIN
Deck 1		
(Compiler, Pass I)		
Card No. 00269,		
Columns 44-48	11200(FHO)	ORIGIN+0(FHO)
Deck 2		
(Compiler, Pass II)		
Card No. 04472,		
Columns 39-43	11200(FHO)	ORIGIN+0(FHO)
Deck 3		
(Library subroutines)		
Card No. 09973,		
Columns 44-48	11200(FHO)	ORIGIN+0(FHO)
Card No. 510266,		
Columns 8-12	11201	ORIGIN+1
Columns 20-24	11205	ORIGIN+5
Columns 32-36	11206	ORIGIN+6
Columns 44-48	11210	ORIGIN+10
Deck 4		
(Library subroutines)		
(For use with the Automatic Floating Point Operations feature)		
Card No. 009973,		
Columns 44-48	11200(FHO)	ORIGIN+0(FHO)
Card No. 510252,		
Columns 8-12	11201	ORIGIN+1
Columns 20-24	11205	ORIGIN+5
Columns 32-36	11206	ORIGIN+6
Columns 44-48	11210	ORIGIN+10

NOTE: FHO indicates that a flag is required in the high-order position of the location specified. ORIGIN is the lowest-numbered core storage location that the programmer desires to use.

IBM 1620 FORTRAN II for Automatic Floating Point

General Description

The 1620 FORTRAN II System for Automatic Floating Point is a modification of the standard 1620 FORTRAN II System described earlier in this publication. The machine configuration for the floating point-oriented system is as follows:

IBM/1620 Data Processing System with 40,000 positions of core storage.
IBM/1622 Card-Read Punch.
Automatic Floating Point Operations feature
Indirect Addressing feature

The significant advantage of the floating point-oriented system is the increased speed of object program execution. This is made possible by the use of in-line arithmetic instructions instead of subroutines for such operations as Floating Add, Floating Subtract, etc.

Since the two systems are very similar, only the differences will be described in this section. All specifications and operating procedures for the standard system are valid for the floating point-oriented system if no specific mention is made of them here.

Language

The language of the two systems is the same except for the addition of two statements in the floating point-oriented system.

IF (OVERFLOW) Statement

This statement causes the program to transfer to a particular statement depending on whether the Arithmetic Overflow indicator is on or off.

General Form

IF (OVERFLOW) n_1, n_2

where n_1 and n_2 are statement numbers. The parentheses enclosing the word OVERFLOW, and the comma separating the statement numbers are required punctuation.

The program transfers to the statement numbered n_1 if the Arithmetic Overflow indicator is on, or to the statement numbered n_2 if it is off. If the indicator is on, it is turned off by the interrogation.

Example

IF (OVERFLOW) 5, 21

means, "If the Arithmetic Overflow indicator is on, transfer to statement 5; otherwise, transfer to statement 21."

IF (EXPONENT CHECK) Statement

This statement causes the program to transfer to a particular statement depending on whether the Exponent Check indicator is on or off.

General Form

IF (EXPONENT CHECK) n_1, n_2

where n_1 and n_2 are statement numbers. The parentheses enclosing the words EXPONENT CHECK, and the comma separating the statement numbers are required punctuation. The space between the words EXPONENT and CHECK is optional.

The program transfers to the statement numbered n_1 if the Exponent Check indicator is on, or to the statement numbered n_2 if it is off. If the indicator is on, it is turned off by the interrogation.

Example

IF (EXPONENT CHECK) 7, 15

means, "If the Exponent Check indicator is on, transfer to statement 7; otherwise, transfer to statement 15."

Arithmetic and Input/Output Subroutines

There is only one subroutine set supplied with this system. The fixed point word precision may vary from 4 to 10 decimal digits. Floating points precision cannot be changed and is set at 8 decimal digits. The control card to vary the fixed point precision is identical to that previously described, except that columns 2 and 3 (which specify floating precision) are ignored. The error message associated with this control card will be

ERROR, K OUTSIDE RANGE

Since the FORTRAN compiler generates in-line arithmetic instructions for such operations as floating add and floating subtract, the arithmetic and input/output subroutine group is reduced to the following:

Floating-Point Arithmetic
- Reverse Subtract
- Reverse Divide
- Set FAC to zero

Fixed-Point Arithmetic
- Reverse Subtract
- Multiply
- Divide
- Reverse Divide

Common Subroutines
- Reverse sign of FAC
- Fix a Floating-Point Number
- Float a Fixed-Point Number

Exponentiation
- Fixed-Point J**I
- Floating-Point A**(±I)
- Floating-Point A**(±B)

Input/output
- Read Card
- Read Tape
- Read Typewriter
- Write Card
- Write Tape
- Write Typewriter

Makeup of Compiler Decks

Pass I

Card Numbers	(Col. 76-80)
00001-00009	Loader
00010-00416	Compiler
02000-02005	Arithmetic Tables
03000	Library Subroutine count and names of library subroutines.

Pass II

Card Numbers	(Col. 76-80)
04001-04009	Loader
04010-04491	Compiler
05000-05005	Arithmetic Tables

Makeup of Subroutine Deck

Card Numbers	*Routine*	*Storage Locations Required*
009970-009991	Loader	None
010001-010019	LOGF	800
020001-020016	EXPF	724
030001-030017	COSF-SINF	820
050001-050016	ATANF	686
060001-060012	SQRTF	512
070001-070003	ABSF	58
510001-510210	Arithmetic and Input/Output subroutines	10000
510211-510221	Modifier cards	None
510222	Trailer card	None

Modification of Origin

To change the origin of an object program, the following modifications must be made.

	STANDARD ORIGIN	MODIFIED ORIGIN
Deck 1 (Processor, Pass I)		
Card No. 00272, Columns 16-20	10000 (FHO)	ORIGIN+0 (FHO)
Deck 2 (Processor, Pass II)		
Card No. 04472, Columns 6-10	10000 (FHO)	ORIGIN+0 (FHO)
Deck 3 (Library Subroutines)		
Card No. 009973, Columns 44-48	10000 (FHO)	ORIGIN+0 (FHO)
Card No. 510221,		
Columns 8-12	10001	ORIGIN+1
Columns 20-24	10005	ORIGIN+5
Columns 32-36	10006	ORIGIN+6
Columns 44-48	10010	ORIGIN+10

NOTE: FHO indicates that a flag is required in the high-order position of the location specified. ORIGIN is the lowest-numbered core storage location that is to be used.

Error Messages

Since most floating instructions appear in-line rather than in a subroutine, object time error messages E1 through E7 have been eliminated. The error conditions previously indicated by these messages can be ascertained by the use of the IF (OVERFLOW) and IF (EXPONENT CHECK) statements. Fixed point overflow can also be detected by this method.

Appendix

Supplementary Information

Restart Procedures

If the operator wants to restart during compilation, he may use the following procedures:

Pass I. Insert and execute a branch to the symbolic address RESTR. After the machine halts, press Start and continue as in Step 7 of OPERATING PROCEDURES (PASS I).

Pass II. Insert and execute a branch to the symbolic address PASS II, and continue as in Step 6 of OPERATING PROCEDURES (PASS II).

Object time. Insert and execute a branch to the indirect address 00409 (49 0040$\bar{9}$).

NOTE: The absolute addresses (actual machine addresses) can be found by referring to the symbolic addresses given in the program listing.

Diagnostic Mode

Pass I of the compiler may be used as a diagnostic pass without punching intermediate output. After the message,

ENTER SOURCE PROGRAM, PRESS START

is typed, the 1620 halts. The operator may then insert the digit 1 (one) into location ERSW and manually branch to location DIAG.

The digit at location ERSW is reset to zero in the initialization process. Therefore, batch processing for diagnostic purposes is not possible without manually resetting the digit at ERSW to 1.

A subprogram run in the diagnostic mode causes a header card to be punched. This card can be ignored.

Storage "Map" of Pass I

After Pass I of the compiler has been loaded and before processing begins, core storage contains the following information:

1. Multiply-Add tables are in locations 00100-00399.
2. Communication area used by Pass II in locations 00401-00483.
3. In the standard system (40,000-digit storage), 350 10-digit fields from 36500 through 39999 are initialized for the symbol table. The first table entries contain the addresses of the numeric representation (two numeric digits for each alpha character) of the relocatable subroutine names. The actual alphameric representation of each name is stored in a packed name table beginning at location 34401.
4. The remaining symbol table entries are initialized to $\bar{0}0400\ddagger000\ddagger$.

If the source machine has 60,000 positions of core storage, the symbol table will be initialized with 800 10-digit fields from 52000 through 59999. In this case the name table will begin at location 40001.

The symbol table is filled up from both ends with constants and statement numbers at the higher addresses, and library subroutine names followed by variables, etc., at the lower addresses.

Storage "Map" of Pass II

After Pass II of the compiler has been loaded, core storage contains the following information:

1. Multiply-Add tables in locations 00100-00399.
2. Communication area from Pass I in locations 00401-00483.
3. The symbol table begins at either 36500 or 52000 depending on whether the program is being compiled on a 40,000 or 60,000 position machine. During batch compilation, the symbol table might not be in memory, but might be read in later at one of the above addresses.

At the end of Pass II, the conditions in memory are the same as above, with the possible exception of additional entries in the symbol table.

If it is necessary for Pass II to generate temporary storage in the decomposition of an arithmetic expression, the entries for the temporary areas are placed in the symbol table, as follows:

$\bar{X}$XXXXM0I00

where $\bar{x}$xxxx is the "object time" address of the temporary storage area.

M is either	(a) 0 (zero) for floating-point mode, or
	(b) 2 for fixed-point mode.
I is either	(a) $\bar{1}$ if the area is in use, or
	(b) 0 if the area is not in use.

M and I may vary during compilation since the same temporary areas can be used many times within one program. Temporary storage areas which are generated during Pass II are placed in line with the object instructions and a branch around the temporary area is supplied by the compiler.

Description of Symbol Table in Storage during Pass I

After the message,

TURN SW 1 ON FOR SYMBOL TABLE, PRESS START

is typed, and before the Start key is pressed, the symbol table entries are in the following forms:

1. *Simple variables*

ĀAAAAMDRRR

where ĀAAAA is	the low-order address of the numeric representation (two numeric digits for each alpha character) of the variable name in the name table.
M is either	(a) 2̄ for a fixed point variable, (b) 0̄ for a floating point variable, (c) 2 for an equivalenced fixed point variable, or (d) 0 for an equivalenced floating point variable.
D is either	(a) 0̄ for a common variable, or (b) 0 for a noncommon variable.
RRR is either	(a) 00‡̄, if the variable was equivalenced and this variable was the base of the equivalence; or if the variable was not equivalenced at all, or (b) a number between 001 and 349 (representing the symbol table entry to which the variable was equivalenced) if the variable was equivalenced.

2. *Dimensioned variables (two entries)*

ĀAAAAMDRRR
AAAAAX̄XXXX

where A, M, and R fields are the same as for simple variables.

D is either	(a) 1 for a 1-dimensional array (b) 2 for a 2-dimensional array (c) 3 for a 3-dimensional array. D is flagged if the array is in Common.
X̄XXXX is either	(a) The total number of elements in the array if the array is not in Common, or (b) the address of the core locations preceding the first element in Common, or (c) an offset reference number if the array was equivalenced.

3. *Floating point constants*

ĀAAAA00C̄C0̄

where ĀAAAA is	the low-order address of the constant's mantissa in the name table.
C̄C is	the characteristic of the constant.

4. *Fixed point constants*

ĀAAAA20CC2̄

where ĀAAAA is	the low-order address of the constant in the name table.
CC is	the length of the constant without counting leading zeros.

5. *Statement numbers*

0̄040̄‡‡N̄NNN

where N̄NNN is the statement number. During compilation, statement numbers have the form of:

0̄040̄1JN̄NNN

where J is either	(a) 0, if the statement is a FORMAT statement. (b) ‡, if the statement ends a DO loop. (c) 1, if neither.
J is	flagged if the statement number is not defined.

6. *Arithmetic statement function names*

ĀAAAAM‡̄ḠG‡̄

where ĀAAAA is	the same as for a simple variable.

M is either	(a) 2̄ for fixed point mode, or (b) 0̄ for floating point mode.
GG is	the number of arguments of the function.

7. *Subroutine and function names*

Ā AAAAM‡00‡̄

where all entries are as described for arithmetic statement function names.

8. *Subprogram name dummy parameters in a Subroutine or Function statement (two entries)*

ĀAAAAMD‡0‡̄
ĀAAAAM000‡̄

where the A, M, and D fields are the same as described for a dimensioned variable. These variables (parameters) are placed in the symbol table twice because they may be dimensioned later.

A 50-digit record in locations 00415 through 00464 indicates any relocatable library subroutines that are to be included in the object program. Starting at 00415, the digit 1 (one) in the record indicates that the corresponding subroutines are required by the object program. The order of the "indicators" is the same as the order of the names of the subroutines read in by the Pass I compiler.

9. *Library function names*

ĀAAAAC̄CCCC

where ĀAAAA is the same as for a simple variable, and C̄CCCC is the core storage address used for linkage to the library subroutine.

Description of Symbol Table after Pass I

After the message,

END OF PASS I

is typed, the symbol table is in the following form:

1. *Simple variables and constants*

X̄XXXMDRR‡

where X̄XXXX is the "object time" address of the variable or constant

M is either	(a) 0 for floating point mode, or (b) 2 for a fixed point mode.
DRR is	the same as it was prior to the storage assignment.

The record mark is used during storage allocation to indicate that the address has been assigned,

2. *Dimensioned variables (two entries)*

X̄XXXXMDĀAA
AB̄BBBC̄CCC0

where X̄XXXX is	an adjusted "object time" address used to compute subscripting.
M is either	(a) 0 for a floating point variable or (b) 2 for a fixed point variable.
D is either	(a) 1 for a 1-dimensional array, (b) 2 for a 2-dimensional array, or (c) 3 for a 3-dimensional array.
ĀAAA is	the first dimension of the array.
B̄BBB is	the second dimension of the array.
C̄CCC is	the third dimension of the array.

If the variable is only a 1-dimensional or a 2-dimensional array, B̄BBB and C̄CCC or C̄CCC alone will be zeros.

3. *Subroutine, function names, and dummy parameters.*

X̄XXXX̄MDRRR

where the X, M, and D fields are the same as those described for simple variables and constants; the R field is the same as it was prior to the storage assignment.

Description of the Symbol Table Listing

If Program Switch 1 is turned on after the message,

TURN SW 1 ON FOR SYMBOL TABLE, PRESS START

is typed, then the "object time" storage addresses of the symbol table will be listed in the following order and form. (No flags will appear if the printer is used to list the symbol table.)

1. *Floating point constants* *Fixed point constants*

M̄MMMMMMME+C̄C X̄XXXX F̄FFFF X̄XXXX

where X̄XXXX is	the low-order address of the constant.
M̄MMMMMMM is	a floating point mantissa.
C̄C is	a floating point exponent.
F̄FFFF is	a fixed point constant.

2. *Simple variables* *Dimensioned variables*

NAME $\bar{X}XXXX$ NAME $\bar{X}XXXX$ $\bar{Y}YYYY$

where $\bar{X}XXXX$	for simple variables is the address at object time where the value for NAME will be stored.
$\bar{X}XXXX$	for dimensioned variables is the address at object time of the first element in the array, NAME.
$\bar{Y}YYYY$	is the address of the last element in the array, NAME.

If NAME* is typed, this indicates a dummy parameter within an arithmetic statement function.

3. *Called subprograms*

NAME $\bar{X}XXXX$

where $\bar{X}XXXX$ is	the location where the starting address of the subprogram will be stored.

Symbol Table Listing for Subprograms

When a subprogram is being compiled, the dummy arguments are listed after statement numbers, as follows:

NAME $\bar{X}XXXX$

where $\bar{X}XXXX$ is the location where the actual address of the variable in the mainline program, corresponding to the argument, NAME, will be stored upon entering the subprogram. The same form is also used for simple and dimensioned variables.

The addresses listed for a subprogram are not the actual addresses at object time. Since subprograms are relocated upon loading, the listed addresses have to be adjusted relative to the starting location of the subprogram.

Statement Number Listing

During Pass II, if Program Switch 1 is on, statement numbers are listed in the form,

$\bar{S}SSS$ $\bar{X}XXXX$

where $\bar{X}XXXX$	is the address of the first instruction generated for statement number $\bar{S}SSS$.

Index

Page

Absolute Value Function 16
ACCEPT Statement 11
ACCEPT TAPE Statement 11
Adding Subroutines 17
Additional Core Storage 24
Alphameric Field Specifications 15
Approximation Methods for Subroutines 17
Arctangent Function 16
Arguments
  of a function 19, 20
  in common storage 13
Arithmetic and Input/Output Subroutines 22
Arithmetic Expressions 4
Arithmetic Modes 2
Arithmetic Precision 3
Arithmetic Statements 5
Arithmetic Statement Functions 18
Arrays 3, 12
  arrangement of, in storage 4
Automatic Fix/Float 10
Automatic Floating Point Feature 37.2

Batch Compilation 27
Blank Field Specification 15

CALL Statement 20, 19
Card Form, FORTRAN 1
Carriage Control 36
Coding Form 1
Comments Statement 1
COMMON Statement 13
  use of COMMON together with EQUIVALENCE 13
Common Storage 13
Compilation Process 22
Compiler Deck
  Non-printer
    Pass I 22
    Pass II 26
  Printer
    Pass I 37
    Pass II 37
Computed GO TO Statement 6
Console Switch Settings 23
Constants
  fixed point 2
  floating point 2
Continuation lines 1
CONTINUE Statement 9
Control Card 24
Control Statements 6
  Computed GO TO 6
  CONTINUE 9
  DO 7
  END 9
  IF 7
  IF (SENSE SWITCH) 7

Page

  PAUSE 9
  STOP 9
  Unconditional GO TO 6
Conversion of Numeric Data 14
Cosine Function 16, 17

Data Input to Object Program 16, 30
DIMENSION Statement 12
DO Statement 7
  DO's within DO's 8
  Exits 9
  Index 8
  Nesting 8
  Range 8
  Restrictions 9
  Preservation of Index Values 8
Dummy Variables 14, 19

E-type Conversion 14, 15, 16
Ending a FORMAT Statement 16
END Statement 9
EQUIVALENCE Statement 12
Error Messages (see Messages)
Execution of Object Program 30
Exponent 2
Exponential Function 16, 17
Expressions 4

F, floating point precision 3, 24
F-type Conversion 14, 15, 16
Field Specifications
  Alphameric 15
  Blank 15
  Numeric 14
Fixed Point
  Constants 2
  Expressions 4
  Variables 2
Fixed to Floating Conversion 10
Floating Point
  Accumulator (FAC) 22
  Arithmetic 2
  Constants 2
  Expressions 4
  Variables 3
Floating to Fixed Conversion 10
FORMAT Statement 14
  ending a, ... 16
FORTRAN Functions 19
FUNCTIONS 16
  Arithmetic Statement 18
  Library 16
  FORTRAN 19
FUNCTION Subprogram 19

GO TO, Computed 6
GO TO, Unconditional 6

Page

Hierarchy of Operations 5

I-type Conversion 14
IF Statement 7
IF (SENSE SWITCH) Statement 7
Index Values, DO Statement 7, 8
Input/Output 10
  in matrix form 10
  of entire matrices 10
Input/Output Statements 10
  ACCEPT 11
  ACCEPT TAPE 11
  FORMAT 14, 36
  PRINT 11, 36
  PUNCH 11
  PUNCH TAPE 11
  READ 10
  TYPE 11
Intermediate Output 24, 26

K, fixed point precision 3, 24

Language 1
Library Functions (subroutines) 16, 30
  adding 34
Linkage 18
Listings
  Printer 36
  Statement Number 41
  Symbol Table 40, 41
Lists
  FORMAT Statement 14, 15, 16
  Input/Output Statement 16
  of quantities, specifying 10
Loading
  Compiler 22, 26
  Subprograms 30
  Subroutines 30
Logarithm Function 16

Machine Configuration iv
Mantissa 2
Matrices 3, 12
  Input/Output of 10
Messages
  Error
    CARD OUT OF SEQUENCE 23, 26, 31, 32
    CARDS NOT IN ORDER 26
    ERROR, F OR K OUTSIDE RANGE 24
    ERROR n (see Table 3) 24
    ERXX (see Table 4) 33
    EXTRA CARD AT END OF DECK 32
    EXTRA HEADER CARD 31
    INSTRUCTION CARD MISSING 31
    LIBRARY FUNCTION DECLARATION ERROR 23
    LIBRARY SUBROUTINE MISSING 27, 32
    MISSING HEADER CARD 27
    OVERLAP $\bar{X}$XXXX DIGITS 33
    OVERLAP $\bar{N}$NNNN 26
    READ SUBPROGRAMS NAMED ABOVE 32
    SSSS + CCCC ERROR n 24
    STATEMENT NUMBER SSSS UNDEFINED 26

Page

    SUBPROGRAM HAS DIFFERENT F OR K 32
    SUBPROGRAM TABLE FULL 31
    SUBROUTINE HEADER MISSING 32
    SUBROUTINE OVERLAP $\bar{X}$XXXX 27
    XXXX DO TABLE FULL 27
    XXXX IMPROPER DO SETTING 27
    XXXX MIXED MODE 27
    XXXX OVERLAP 26
    XXXX SYMBOL TABLE FULL 27
  Instructions to operator
    END OF PASS I 24
    END OF PASS II 26
    ENTER DATA 30
    ENTER SOURCE PROGRAM, PRESS START 22, 27
    LOAD SUBROUTINES 30
    SW1 ON TO PUNCH SUBROUTINES, PRESS START 26
    TURN SW1 ON FOR SYMBOL TABLE, PRESS START 24
    XXXXX LENGTH 27
    YYYYY NEXT COMMON 27
Mode, Diagnostic 38
Modes, Arithmetic 2
Modification of
  compiler deck 34
  library subroutine deck 34
  origin
    non-printer 35
    printer 37
Multiple Record Formats 16

Naming Functions 18,19
Nesting DO's 8
Numeric Field Specifications 14

Object Machine 24
Object Program 29
  loading of 30
  execution of 30
Operating Procedures 22
Overflow Switch 22, 26, 30

Page Heading Card 24
Parentheses 5
Pass I Procedures 22
Pass II Procedures 26
PAUSE Statement 9
Preservation of Index Values 8
Printing Multiple Lines 16
Printer-oriented system 36
PRINT Statement 36
PUNCH Statement 11
PUNCH TAPE Statement 11

READ Statement 10
Repetition of Field Format 15
Repetition of Groups 15
Restart Procedures 38
RETURN Statement 21, 19
Rules for Constructing Expressions 4

Scale Factors 15
Sense Switch Settings 23
Sine Function 16, 17
Skipping, printer 36

Page

Source Program
  Control Card 24
  entering into computer 24
  errors in 24
  statements 1
Spacing, printer 36
Specification Statements 12
  DIMENSION 12
  EQUIVALENCE 12
  COMMON 13
Square Root Function 16, 17
Statements
  Arithmetic 5
  Control 6
  Input/Output 10
  Specification 12
STOP Statement 9
Storage Maps 38, 39
Subprograms
  FUNCTION 19
  SUBROUTINE 20, 19
  loading of 30
Subroutines
  adding 34
  error checks 27, 33
  list of 16, 22
  loading of
    compile time 26
    object time 32
  versions 35
SUBROUTINE Statement 20, 19
Subscripted variables 4
Subscripts 3
Switch settings 23
Symbol Table
  description of 39, 40
  listing 40

Table, Symbol 24, 39, 40
Trace Feature 31
Transfer of Control in a DO 8
TYPE Statement 11
Typewriter
  input 24
  output 11

Unconditional GO TO 6

Variables
  fixed point 2
  floating point 3
  subscripted 4

Work areas 18
Writing library functions 35

C26-5876-2

IBM 1620 Printed in U.S.A. C26-5876-2

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]